A logic query $Q$ is a triple $\langle G, LP, D \rangle$, 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 $LP \cup D$ 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 $\langle G, LP, D \rangle$, where $G$ is the
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 Pl.

\[
\begin{align*}
      r_0: & \ g(X,Y) : - \ up(X,W), \ down(Z,Y), \ g(W,Z). \\
      r_1: & \ g(X,Y) : - \ flat(X,Y). \\
      r_2: & \ up(a,a_1). \\
      r_3: & \ up(a_1,a_2). \\
      r_4: & \ up(a_1,a_3). \\
      r_5: & \ up(a_4,a_2). \\
      r_6: & \ up(a_5,a_4). \\
\end{align*}
\]
METHODS FOR LOGIC-QUERY IMPLEMENTATION

This logic program is a syntactic variation of the well-known same-generation example. In fact, the predicates \textit{up}, \textit{flat}, and \textit{down} correspond to the relations \textit{Child}, \textit{Sibling}, and \textit{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 \(P_1\) if either the fact \(\text{flat}(x, y)\) is in \(P_1\) 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

\[
\text{up}(x, w_1); \text{up}(w_1, w_2); \ldots; \text{up}(w_{i-1}, w_i), \text{flat}(w_i, z_i);
\]

\[
\text{down}(z_i, z_{i-1}); \ldots; \text{down}(z_2, z_1); \text{down}(z_1, y)
\]

are in \(P_1\).

As an example, consider the two constants \(a\) and \(b_3\). The fact \(g(a, b_3)\) can be inferred from \(P_1\) in two different ways:

(1) \(\text{up}(a, a_1); \text{flat}(a_1, b_1); \text{down}(b_1, b_3)\), or

(2) \(\text{up}(a, a_1); \text{up}(a_1, a_2); \text{flat}(a_2, b_1); \text{down}(b_1, b_2); \text{down}(b_2, b_3)\).

The logic program \(P_1\) 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) \(A_u = \{(x, y) | \text{up}(x, y) \text{ is in } P_1\}\);
(2) \(A_f = \{(x, y) | \text{flat}(x, y) \text{ is in } P_1\}\);
(3) \(A_d = \{(x, y) | \text{down}(x, y) \text{ is in } P_1\}\).

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.](image-url)
Using the graphical representation, it is easy to see that a fact $g(x, y)$ can be inferred by $P_1$ 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 $P_1$ 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 $\text{?} g(u, Y)$. 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 is 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:

$$f(g) = \text{flat} \cup \pi_{1,6}(\text{up} \uparrow_{2=1} g) \uparrow_{4=1} \text{down}.)$$

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

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

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 $p(x)$, where $x$ 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 $DG_L = (S, A)$ 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 $g$, a rule $r$ in $L$ defines $g$ if the head predicate symbol of $r$ is $g$.

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 $\langle G, LP_G, D \rangle$; 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 $LP \cup D$ and unify with $G$.

Example 1. Consider the same-generation example of the previous section (Program P1). A possible logic query is $\langle g(a, Y), LP, D \rangle$, where $LP = \{r_0, r_1\}$ 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.

---

¹The 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 $Q = \langle G, LP, D \rangle$ 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 $Q = \langle G, LP, D \rangle$ 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 $Q = \langle g(a, Y), LP_1, D \rangle$, where $LP_1$ is the logic program

\begin{align*}
   r_0 &: g(X, Y) \leftarrow b(X, W), g(W, Z), g(U, V), c(V, Y). \\
   r_1 &: g(Z, W) \leftarrow d(Z, W).
\end{align*}

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 $LP_2$:

\begin{align*}
   r_0 &: g(X, Y) \leftarrow g(X, Z), c(Z, Y). \\
   r_1 &: g(Z, W) \leftarrow b(Z, Y), g(Y, W). \\
   r_2 &: g(U, V) \leftarrow d(U, V).
\end{align*}

then the query $\langle g(a, Y), LP_2, D \rangle$ is linear but not strongly linear. On the other side, given the following logic program $LP_3$:

\begin{align*}
   r_0 &: g(X, Y) \leftarrow p(X, Z), c(Z, Y). \\
   r_1 &: p(Z, W) \leftarrow b(Z, Y), g(Y, W). \\
   r_2 &: p(U, V) \leftarrow d(U, V).
\end{align*}

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

\begin{align*}
   r_0 &: g(X, Y) \leftarrow b(X, W), g(W, Z), c(Z, Y). \\
   r_1 &: g(U, V) \leftarrow d(U, Z), c(Z, V).
\end{align*}

It is easy to see that $\langle g(a, Y), LP_3, D \rangle$ and $\langle g(a, Y), LP_4, D \rangle$ have the same answers. Finally, the query in Example 1 is a CSL query.

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

\begin{align*}
   r_0 &: g(x_0) \leftarrow C_0, r(y). \\
   r_1 &: g(x_1) \leftarrow C_1. \\
   \vdots \\
   r_{lp} &: g(x_{lp}) \leftarrow C_{lp}.
\end{align*}
where \( x_0, \ldots, x_n, y \) are lists of arguments, \( \ell p = |\ell P| - 1 \), 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 \( x \) be distinct and that \( x_0 \) and \( y \) 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 \( g(x_0) \) 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 \( y \) 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 \( Q = \langle g(x), LP, D \rangle \) 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 \( g(x) \), 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 \( \langle g(a, b, Z, W), LP, D \rangle \), where \( D \) contains a number of facts defining the database predicates \( a, b, c, d, e, f \), and where \( LP \) is

\[
\begin{align*}
    r_0: & \ g(X, Y, Z, W) :- a(X, U), b(Y, U, Z), c(Z, X, \bar{Y}), \\
    & \ d(W), e(\bar{W}), g(\bar{X}, \bar{Y}, \bar{Z}, \bar{W}). \\
    r_1: & \ g(X, Y, Z, W) :- f(X, Y, Z, W).
\end{align*}
\]

Suppose that \( S = \{1, 2\} \); then the database predicates that are bound by \( S \) are \( a(X, U) \) and \( b(Y, U, Z) \). Moreover, \( T^S = \{3\} \); thus the argument \( Z \) of \( g(X, Y, Z, W) \) 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 \( Q = \langle G, LP, D \rangle \) 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 \( O(2^n|LP|) \), where \( |LP| \) 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 \( S_i, 1 \leq i \leq k \), are the nodes of \( B_Q \); \( S_1 \) is the source node; the arcs \( (S_i, S_{i+1}) \), \( 1 \leq i \leq j-1 \), 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.
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 $\langle g(a, b, Z, W), I.P, D \rangle$ of Example 3. We have $L^S_1 = a(X, U), b(Y, U, Z)$ and $L^S_2 = c(Z, X, Y)$, where $S_1 = \langle 1, 2, 3 \rangle$ and $S_2 = \langle 3 \rangle$. In addition, $S^S_1 = \langle 3, 4 \rangle$ and $L^S_{S_1}$ is equal to $c(Z, X, Y), d(W)$, $e(\hat{W})$. Finally, $S^S_2 = \langle 1, 2, 4 \rangle$ and $L^S_{S_2}$ is equal to $a(X, U), b(Y, U, Z), d(W)$, $e(\hat{W})$.

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

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

Example 5. If $z = \langle X, Y, Z, W \rangle$ and $U = \langle 1, 2, 4 \rangle$, then $z(U) = \langle X, Y, W \rangle$. Let $\sigma = \langle (X, a), (Y, b), (Z, c), (W, a) \rangle$ and $L = p(X, d, Y), q(Y, W)$. Then $z(\sigma) = \langle a, b, c, a \rangle$ and $L^\sigma = p(a, d, b), q(b, a)$. Finally, given the binding graph $\langle S_1, S_2, S_3, S_4 \rangle$, we have $S^S_1 = S_2$, $S^S_2 = S_3$, and $S^S_3 = S_2$.

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 $G_Q = (N_u \cup N_d, A_u \cup A_f \cup A_d)$ 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 $[S_1, a]$ is in $N_u$ (source node), where $a$ is the list of constants in the query goal.\(^3\)

(b) If $[S_i, a]$ is in $N_u$ and there exists a substitution $\sigma$ for the set of variables in $L^S_i$ and $x_0(S)$ such that $x_0(S)\sigma = a_1$ and every predicate in $L^S_i\sigma$ is in $D$, then the node $[S_{i+}, a_2]$ is in $N_u$ and the arc $([S_i, a_1], [S_{i+}, a_2])$ is in $A_u$.

\(^3\)Here and in what follows, constants are listed in the order in which they appear in the predicate.
METHODS FOR LOGIC-QUERY IMPLEMENTATION

343

If \([S_i, a_1]\) is in \(N_u\) and there exists both a nonrecursive rule in LP, say

\[ r_q: g(x_a) \leftarrow C\sigma, \]

and a substitution \(\sigma\) for the set of variables in \(C_q\) and in \(x_a\) such that

\[ x_a(S_i)\sigma = a_1 \]

and every predicate in \(C\sigma\) is in \(D\), then the node \([S_i, b_1]\) is in \(N_d\) and the arc \((S_i, a_1), (S_i, b_1)\) is in \(A_f\), where \(b_1 = x_q(S_i)\sigma\).

d) If \([S_q, b_2]\) is in \(N_d\), there is \(S_i\) such that \(S_i+ = S_q\), and there exists a substitution \(\sigma\) for the set of variables in \(L_{S_i}\) in \(x(S_i)\), and in \(x(S_i)\sigma\) such that \(x(S_i)\sigma = b_2\) and every predicate in \(L_{S_i}\sigma\) is in \(D\), then the node \([S_i, b_1]\) is in \(N_d\) and the arc \((S_i, b_2), (S_i, b_1)\) is in \(A_d\), where \(b_1 = x_q(S_i)\sigma\).

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

<table>
<thead>
<tr>
<th>a</th>
<th>e1</th>
<th>b</th>
<th>e1</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>e2</td>
<td>b</td>
<td>e1</td>
<td>c2</td>
</tr>
<tr>
<td>a</td>
<td>e3</td>
<td>b</td>
<td>e2</td>
<td>c3</td>
</tr>
<tr>
<td>a</td>
<td>e3</td>
<td>b</td>
<td>e3</td>
<td>f1</td>
</tr>
</tbody>
</table>

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 arc 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 \(\langle g(a, Y), LP, D\rangle\), 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.
call these arguments an answer tuple. More formally, an answer tuple of Q is a tuple b of constants such that there exists an answer g(c) for which e(Si−) = b, where Si− 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 ≥ 0, from a node in Nu to a node in Nd such that the first i arcs are in Au, the (i + 1)th arc is in Af, and the last i arcs are in Ad.

Theorem 1. Let Q = (G, LP, D) be a bound CSL query, and GQ be the query graph of Q. If a tuple b is an answer of Q, then there exists an answer path from the source node [Si, a] to the node [Si−, b] in GQ.

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 [Si, a] to a node [Si−, b] in GQ, then the tuple b is an answer of Q. Unfortunately, this is not the case, as shown in the next example.

Example 7. Consider the query (g(X, a), LP, D), where LP is

\[ g(X, Y) : - p_1(X, \hat{X}, Y), g(\hat{X}, \hat{Y}), p_2(\hat{X}). \]
\[ g(X, Y) : - p_3(X, Y). \]
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 $[(2), (a)]$, and the Herbrand universe is $\{a, a_1, a_2, a_3\}$. Consider the substitution $\sigma = ((X, a_2), (Y, a), (\bar{Y}, a_1))$ for the variables in $L^{(1)} = p_1(X, \bar{Y}, Y)$ 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 $[(2), \langle a_1 \rangle]$ is in $N_u$ and the arc $([2], [a])$ 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 $[(2), \langle a_1 \rangle]$ is in $N_u$ and $p_3(a_3, a_1)$ is in $D$, by the definition of query graph [part (c)] the node $[(1), \langle a_1 \rangle]$ is in $N_d$ and the node $[(1), \langle a_1 \rangle]$ is in $A_f$ (and not other arc is in $A_f$). Finally, consider the following four different substitutions for the variables in $L^{(1)} = p_2(\hat{X})$, in $\mathbf{y}(\langle 1 \rangle) = \Pi \hat{X}$ and in $\mathbf{x}_0(\langle 1 \rangle) = \langle X \rangle$ (note that $S^- = \langle 1 \rangle$):

- $\sigma_1 = \{(X, a), (\hat{X}, a_3)\}$.
- $\sigma_2 = \{(X, a_1), (\hat{X}, a_3)\}$.
- $\sigma_3 = \{(X, a_2), (\hat{X}, a_3)\}$.
- $\sigma_4 = \{(X, a_3), (\hat{X}, a_3)\}$.

Since the fact $p_3(a_3)$ is in $D$, by the definition of query graph [part (d)] the nodes $[(1), \langle a \rangle], [(1), \langle a_1 \rangle], [(1), \langle a_2 \rangle]$ are in $N_d$ (besides the node $[(1), \langle a_3 \rangle]$.

![Figure 4. Graphs for the query of Example 7: (a) binding graph, (b) query graph.](image-url)
that is already in $N_d$), and the arcs $([1, \langle a_2 \rangle], [1, \langle a_3 \rangle])$, $([1, \langle a_3 \rangle], [1, \langle a_1 \rangle])$ and $([1, \langle a_3 \rangle], [1, \langle a_2 \rangle])$ 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 $[1, \langle a \rangle]$ to $[1, \langle a \rangle]$, $[1, \langle a \rangle]$, and $[1, \langle a \rangle]$ does not guarantee that $g(a, a)$, $g(a_1, a)$, and $g(a_2, a)$ 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 $Q = \langle G, LP, D \rangle$ be a bound CSL query and $B_Q$ be its binding graph. $Q$ is 1-bound if for each node $S$ in $B_Q$,

$$X \cap B_S = \emptyset,$$

where $B_S$ is the set of all variables bound by $S$, and $X$ 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 $S_1 = [1]$, is in $B_2$, where $S_2 = [2]$ is a node of the binding graph [see Figure 4(a)]. On the other hand, the query $\langle g(a, b, Z, W), LP, D \rangle$ 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 $\langle g(a, Y), LP_1, D \rangle$, where $D$ is any set of facts, is not 1-bound, since the variable $Z$ is both in $L^{S_1}$ and in $L^{S_2}$ [note that $S_1 = [1]$, $S_1 = [2]$, $L^{(1)}$ is $a(X, Z)$, and $L^{(2)}$ is $b(W, Y)$, $Z > Y]$.

Theorem 2. Let $Q = \langle G, LP, D \rangle$ 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 $[S_1, a]$ to a node $[U, b]$ in $N_d$, then the tuple $b$ 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 $Q = \langle G, LP, D \rangle$, if $Q$ is not 1-bound, then there exists a query $Q = \langle G, LP, D \rangle$ such that there is an answer path from the source node $[S_1, a]$ to a node $[S_1, b]$ in the query graph of $Q$, and $b$ is not an answer tuple of $Q$. 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.

---

4This 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 $Q = \langle G, L, P, D \rangle$ 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 $G_Q$ is composed of three subgraphs: $G_u = (N_u, A_u)$, $G_f = (N_f, A_f)$, and $G_d = (N_d, A_d)$. Furthermore, $N = N_u \cup N_f$, $N_u$ 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 $G_u$ 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 $\text{adj}_G(X)$ the set of all nodes $j$ such that the arc $(i, j)$ in $G$ and $i$ is in $X$. In other words, $\text{adj}_G(X)$ is the set of all nodes that are adjacent to some node in $X$, whereas $\text{adj}^{-1}_G(X)$ 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 $\langle g(a, b, Z, W), L, P, D \rangle$ 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

$\text{adj}_G(X) = \{\langle 1, 2 \rangle, \langle a_1, b_1 \rangle\}$,

$\text{adj}^{-1}_G(X) = \{\langle 1, 2 \rangle, \langle a, b \rangle\}$,

and

$\text{adj}_G(X) = \{\langle 1, 2, 4 \rangle, \langle h_1, g_1, l_1 \rangle\}$.

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

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

computes $\text{adj}_G(X)$. 
\[ U := \{a\}; \]
\[ \text{Answer} := \text{adj} G_f(U); \{\text{INITIALIZATION}\} \]
\[ v := 0; \]
\[ \text{While } U \neq \emptyset \text{ do} \]
\[ \begin{align*}
U &:= \text{adj} G_f(U); \; v := v + 1; \{\text{UP}\} \\
D &:= \text{adj} G_f(U); \{\text{FLAT}\} \\
t &:= v; \\
\text{While } D \neq \emptyset \text{ and } t > 0 \\
&\text{ do begin} \\
&\quad D := \text{adj} G_f(D); \{\text{DOWN}\} \\
&\quad t := t - 1
\end{align*} \]
\[ \text{end}; \]
\[ \text{Answer} := D \cup \text{Answer} \]

\begin{figure}[h]
\centering
\begin{minipage}[t]{0.5\textwidth}
\begin{algorithmic}
\STATE \textbf{While } U \neq \emptyset \\
\quad \textbf{do} \begin{align*}
U &:= \text{adj} G_f(U); \; v := v + 1; \{\text{UP}\} \\
D &:= \text{adj} G_f(U); \{\text{FLAT}\} \\
t &:= v; \\
\textbf{While } D \neq \emptyset \text{ and } t > 0 \\
&\textbf{do begin} \\
&\quad D := \text{adj} G_f(D); \{\text{DOWN}\} \\
&\quad t := t - 1
\end{align*} \\
\textbf{end}; \\
\textbf{end}
\end{algorithmic}
\caption{Eager method.}
\end{minipage}
\end{figure}

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. 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.
METHODS FOR LOGIC-QUERY IMPLEMENTATION

U_0 := \{a\}; v := 0;
{1st PHASE: UP}
While \( U_v \neq \emptyset \) do
begin
\( U_{v+1} := \text{adj} G_0(U_v); v := v + 1; \)
end;
D_{v-1} := \text{adj} G_f(U_{v-1});
{2nd PHASE: FLAT and DOWN}
For \( i := v - 1 \) downto 1 do
\( D_{v-1} := \text{adj} G_d(D_v) \cup \text{adj} G_f(U_{v-1}); \)
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 \((i, \hat{i})\) such that \((i, \hat{i})\) is in \( A_u \) and \((\hat{j}, j)\) 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 \( i \) 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
\( \hat{N}_u := \text{adj} G_u(\hat{N}_u) - N_u; \)
\( N_u := \hat{N}_u \cup N_u; \)
end;
{COMPUTING THE ANSWER PATHS}
\( \hat{E} := \{(i, j)\}((i, j) \in A_f \text{ and } i \in N_u); (a) \)
\( sE = \hat{E}; \)
While \( \hat{E} \neq \emptyset \) do
begin
\( \hat{E} := \{(i, \hat{i})\}((i, \hat{i}) \in A_u, (\hat{i}, j) \in E, (\hat{j}, j) \in A_d) - \hat{E}; (b) \)
\( E = E \cup \hat{E}; \)
end;
Answer := \{(j \mid (a, j) \in E)\}.

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

We note that the computation of additional pairs $\bar{E}$ 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 $\bar{E}$. We stress that such adjacent nodes are in general found using relational-algebra operations on the database [7]. This means that a pair $(i, i')$ 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 $\bar{E}$.

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).
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.
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 $v$), 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 $O(n_n m_n)$. 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.](image_url)
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 \(O(m_r)\). 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 \(O(m_u + m_r)\) time, since each arc is considered once. In all other cases, for each pair \((i, j)\) in \(E\), 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
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 \( x \in N_o \), 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 \( R(d(x)) \) 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. \( \square \)

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 \( 2n_un_d + 1 \).

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 \( i \), \( 1 \leq i \leq k - 1 \), 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 $i, j$, $1 \leq i < j \leq k$, such that $a_i = a_j$ and $b_i = b_j$. Consider now the following sequence of nodes:

$$<a_1, \ldots, a_{i-1}, a_i, a_{j+1}, \ldots, a_k, b_k, \ldots, b_{j+1}, b_j, b_{i-1}, \ldots, b_1>.$$ 

We have that, by assumption, $(a_k, b_k)$ is in $A_f$ and for each $h$, $1 \leq h \leq i - 1$ and $j + 1 \leq h < k - 1$, $(a_h, a_{h+1})$ and $(b_h, b_{h+1})$ are in $A_u$ and in $A_d$, respectively. On the other hand, $(a_j, a_{j+1})$ is in $A_u$ and $(b_{j+1}, b_j)$ 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 $u \geq 0$. 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 $D_v$, $u > v \geq 0$, was already used for the previous step $k$, $v \leq k < u$, 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 := $\emptyset$;
$U_0 := \{a\}$; $u := 0$;
$N_u := \emptyset$; $N_d := \emptyset$;

Repeat

$N_u := N_u \cup U_u$;
$D_u := \text{adj} G_f(U_u)$;
$\tilde{D}_u := D_u$;

For i := u downto 1 do

begin
$\tilde{D}_{i-1} := \text{adj} G_d(\tilde{D}_i) - D_{i-1}$;
$D_{i-1} := \tilde{D}_{i-1} \cup \tilde{D}_{i-1}$;
$N_d := N_d \cup \tilde{D}_{i-1}$;
end;

$u := u + 1$;
$U_u := \text{adj} G_d(U_{u-1})$

until ($U_u = \emptyset$) or ($u > |N_u| \times |N_d|$)

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 $j \leq p$. 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 $q(a)$, where $a$ is a list of bound arguments. A derivation tree for $q(a)$ 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 $\sigma$ for all variables in $r$ such that $P\sigma = N_i$ ($1 \leq i \leq m$). The substitution $\sigma$ is called a solving substitution for $N$. The root of the derivation tree is $q(a)$.

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

**Lemma 1.** Let $Q = (G, LP, D)$ 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 $n_1 = [S_i, a_i]$ and $n_3 = [U, a_2]$; then $U = S_i a_i$.

**PROOF.** Straightforward. □

**Theorem 1.** Let $Q = (G, LP, D)$ 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 $[S_i, a_i]$ to the node $[S_i, b_i]$ in $G_Q$.

**PROOF.** In order to prove the theorem, it is sufficient to show that, given any node $[S_i, a_i]$ in $N_u$, if the fact $g(c_i)$ is inferred from $LP \cup D$, where $c_i(S_i) = a_i$, then there is an answer path from $[S_i, a_i]$ to $[S_i, b_i]$ in $G_Q$, where $c_i(S_i) = b_i$. 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 $g(c_i)$. We denote this number by $s$; furthermore, let $\sigma$ denote the solving substitution for the root of DT.

**Basis of the induction** ($s = 1$). Then the root $g(c_i)$ of DT is labeled by a nonrecursive rule, say

$$r_q: g(x_q) = C_q,$$

and the children of the root are the facts $P\sigma$ in $D$ corresponding to predicates $P$ in $C_q$. We have that $x_q(S_i)\sigma = a_i$ and $x_q(S_i)\sigma = b_i$. Therefore, since $[S_i, a_i]$ is in $N_u$ by assumption, the node $[S_i, b_i]$ is in $N_d$ and the arc $([S_i, a_i], [S_i, b_i])$ 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 \( g(c_1) \) of DT is labeled by the recursive rule, and its children are the fact \( g(y)\sigma \) and the facts \( P\sigma \) corresponding to the datum predicates \( P \) in the recursive rule. Therefore, for every predicate \( P \) in \( L^s \) or in \( L^{S_i} \), \( P\sigma \) is in \( D \). Say that \( g(y)\sigma = g(c_2) \), where \( c_2(S_\alpha) = a_2 \) and \( c_2(S_\alpha) = b_2 \). We have that \( x_0(S)\sigma = a_1 \), \( y(S_\alpha)\sigma = a_2 \), \( P\sigma \) is in \( D \) for every predicate \( P \) in \( L^s \), and \([S_\alpha, a_1]\) is in \( N_u \) by assumption. Hence, the arc \([S_\alpha, a_1], [S_\alpha, a_2]\) 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 \( g(c_2) \) is a derivation tree with \( s - 1 \) recursive nodes, by the inductive hypothesis there is an answer path from \([S_\alpha, a_2]\) to \([S_\alpha, b_2]\). Consider now the node \([S_\alpha, b_2]\). Obviously, this node is in \( N_d \); moreover, we also have that \( x_0(S)\sigma = b_1 \), \( y(S_\alpha)\sigma = b_2 \), 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 \([S_\alpha, b_2], [S_\alpha, b_2]\) is in \( G_Q \). It follows that, by Lemma 1(a), there is an answer path from \([S_\alpha, a_1]\) to \([S_\alpha, b_1]\).

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 \([S_1, a_1]\) in \( N_u \) to a node \([U, b_1]\) in \( N_d \), then \( U = S_j^\alpha \).

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 \((S_j, S_i) \) and \((S_k, S_j) \). By definition of binding graph, the set of variables bound by \( S_j \) in \( r_0 \) equals the set of variables bound by \( S_k \) in \( r_0 \); thus \( B_{S_j} = B_{S_k} \). Since \( S_j \neq S_k \), there exists an argument in the head predicate of \( r_0 \) that is denoted by an index of \( S_j \) 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 \( B_{S_j} \). But \( B_{S_j} = B_{S_k} \); 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 \( n_1 = [S_j, a_1] \) and entering \( n_2 = [U, b_1] \). If \( s \) is 1, then obviously \( U = S_j^\alpha \). 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, \( n_3 = [S_{j - 1}, a_2] \). Therefore, by the inductive hypothesis, \( n_3 = [S_{j - 1}^\alpha, b_2] \). By Lemma 2(a) there is only one node \( S_u \) such that \( S_u^\alpha = S_{j - 1}^\alpha \). Hence, \( U = S_j^\alpha \).

Theorem 2. Let \( Q = \langle G, LP, D \rangle \) 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 \([S_1, a]\) to a node \([U, b]\) in \( N_u \), then the tuple \( b \) is an answer tuple of \( Q \).

Proof. By Lemma 2(b), \( U = S_j^\alpha \). In order to prove the theorem, it is sufficient to show that, given any node \([S_1, a_1]\) in \( N_u \), if there is an answer path from this node to a node \([S_j^\alpha, b_1]\) in \( N_d \), say with length \( 2s + 1 \), then the fact \( g(c_1) \) is inferred from
METHODS FOR LOGIC-QUERY IMPLEMENTATION

LP ∪ D, where c_(Si) = a_1 and c_(Si^+) = b_1. We carry out this proof by induction on s.

Basis of the induction (s = 0). The arc ([S_i, a_1], [S_i^-, 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

\[ r_q: g(x_q) \leftarrow C_q, \]

and a substitution σ such that x_q(S_i) = a_1, x_q(S_i^-) = b_1, and Pσ is in D for every P in C_q. It follows that a derivation tree for g(c_1) can be easily constructed. Therefore, g(c_i) is inferred from LP ∪ 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, a_2] and [T, b_2], such that the arcs ([S_i, a_1], [S_i^+, a_2]) and ([T, b_2], [T, b_1]) are in A_f and A_d, respectively, and there is an answer path from [S_i^+, a_2] to [T, b_1] with length 2(s - 1) + 1. By Lemma 2(b), T = S_i^+. By the inductive hypothesis, the fact g(c_2), where c_2(S_i^+) = a_2 and c_2(S_i^-) = b_2, is inferred from LP ∪ D. We now construct a derivation tree DT for g(c_i) as follows. The root is obviously g(c_i) and is labeled by the recursive rule r_o. A child of the root is g(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(c_2), since this fact is inferred from LP ∪ 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 x_0(S_i)σ_1 = a_1, y(S_i^+)σ_1 = a_2, Pσ_1 is in D for every P in L^S_i, x_0(S_i^-)σ_2 = b_1, y(S_i^-)σ_2 = b_2, and Qσ_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 Pσ_1 for every P in L^S_i and a child Qσ_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(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 ∪ V_2 and V_1 = B_s, where B_s 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, or, therefore, in V_1. Hence, V_1 ∩ V_2 = ∅. Therefore, we can set σ(V_1) = σ_1 and σ(V_2) = σ_2. 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


