MAGIC SETS AND OTHER STRANGE WAYS TO IMPLEMENT LOGIC PROGRAMS

(Extended Abstract)

Francois Bancilhon
MCC
David Maier[1]
Oregon Graduate Center
Yehoshua Sagiv[2]
Jeffrey D. Ullman[3]
Stanford University

ABSTRACT

Several methods for implementing database queries expressed as logical rules are given and they are compared for efficiency. One method, called "magic sets," is a general algorithm for rewriting logical rules so that they may be implemented bottom-up (= forward chaining) in a way that cuts down on the irrelevant facts that are generated. The advantage of this scheme is that by working bottom-up, we can take advantage of efficient methods for doing massive joins. Two other methods are ad hoc ways of implementing "linear" rules, i.e., rules where at most one predicate in any body is recursive. These methods are introduced not only because they could be the right way to implement certain queries, but because they illustrate the difficulty of proving anything concrete about optimal ways to evaluate queries.

[1]Work supported by NSF grant IST–83–51730, cosponsored by Tektronix Foundation, Intel, Mentor Graphics, DEC, Servio Logic Corp., IBM, Xerox and Beaverton Chamber of Commerce.

[2]On a leave of absence from Hebrew University. Work supported by a grant of AT&T Foundation and a grant of IBM Corp.

[3]Work supported by NSF grant IST–84–12791 and a grant of IBM Corp.

I. Introduction

We assume that the reader is familiar with the notion of a logic program and with the notation of Prolog. We follow Prolog notation, although we do not assume the left-to-right execution of literals that is inherent in that language. We also assume familiarity with the notation of rule/goal graphs defined in Ullman [1985]. In this paper we deal with *Datalog*, as in Maier and Warren [1985]. In Datalog, literals have no function symbols within them, so we have a greatly simplified subcase of logic programs, but one that is very significant in the logic-and-databases school of thought, e.g., as expressed in Gallaire and Minker [1978]. While all Datalog programs have finite minimum models, that finiteness is no guarantee of convergence of evaluation methods in the presence of recursion. In fact, Brough and Walker [1984] have shown that a top-down, left-to-right Datalog interpreter that only examines ancestor goals cannot be both complete and convergent. We also designate some predicates as database relations. The only clauses for such predicates are ground facts. We expect that a database relation predicate will be stored as a list of tuples in an underlying database system.

Example 1: Let us introduce the logic program for "cousins at the same generation," which will serve as a running example. It is a canonical example of a *linear* logic program. The rules in such programs have at most one occurrence of a recursive predicate, *sg* (same generation) in this case, in any body. The rules are:

r_1: $sg(X,X)$.
r_2: $sg(X,Y)$:- $par(X,X1)$, $par(Y,Y1)$, $sg(X1,Y1)$.

We assume that *par* is a database relation, and $par(U,V)$ means that V is a parent of U. The intention of $sg(U,V)$ is that U and V are cousins, i.e., they have a common ancestor W, and there are lines of descent from W to U and V covering the same number of generations. Rule r_1 says that anyone is his own cousin, i.e., $U = V = W$ and the descent is through zero generations. The only other way for X and Y to be cousins is expressed by r_2, i.e., there are parents of X and Y that are cousins.

We do not assume that the parenthood relationship is well organized by levels. It is possible that someone married his own granddaughter. Thus, we may not assume that the number of generations between two individuals is unique. In fact, we shall not even assume that the parenthood relation is acyclic. As we shall see, the proper algorithm to apply to these rules may depend on whether acyclicity of parenthood may be assumed. □

Let us focus on the query $sg(a,W)$, i.e., find all the cousins of a particular individual a. There are two obvious but expensive ways to implement the rules of Example 1. The first, top-down, or "backward chaining," is the one Prolog would do. Actually, Prolog would run forever on r_1 and r_2, but we can fix the problem by reordering goals, replacing r_2 by

r_2': $sg(X,Y)$:- $par(X,X1)$, $sg(X1,Y1)$, $par(Y,Y1)$.

However, given goal $sg(a,W)$, the Prolog program would consider each parent of a, say b and c. Recursively, it would find all b's cousins, then all c's cousins, and find all the children of both. As b and c may have many ancestors in common, we shall repeat much work finding those ancestors, and many individuals may be cousins of both. Thus, it should not surprise one that the total running time of the Prolog program may be exponential in the number of individuals in the database. The flaw with the naive application of backward chaining, as in Prolog, is that it seeks to discover "all proofs" rather than just "all answers." Its use is more appropriate when only a single answer is desired, in which case finding one proof suffices for generating one answer. Our anticipated applications are biased towards all answers, and we want to avoid finding all proofs of those answers, if possible.

The bottom-up, or "forward chaining" approach can also be very expensive. Here, we start by assuming only the facts in the database, i.e., the *par* relation. The initial estimate for nondatabase relations, *sg* in this example, is the empty set. We then apply the rules, substituting our current estimate for each relation in the bodies and computing, using the natural join and union, a relation for each predicate symbol appearing on the left side of one or more rules. We add the newly generated facts to the current estimate. (For our example, it suffices to use the newly created facts as the next estimate, accumulating the final answer off to the side.) In our example, the first time through, r_2 yields nothing, because sg has no tuples yet. However, r_1 yields the set of all (X, X) pairs. It is not completely clear how the domain of X should be limited, but as there are no function symbols in our rules, it is always safe to limit a variable to the set of all values appearing in the database.

On the second pass, rule r_2 now gives us all facts $sg(U, V)$ where U and V have a parent in common. On the third pass we get all facts $sg(U, V)$ where U and V have a grandparent in common, and so on. This process must converge in time that is polynomial in the number of individuals in the database, whereupon we can restrict the relation to those pairs whose first component is a, and extract the answer to our query $sg(a, W)$.

While the polynomial time bound seems better than the exponential bound for the top-down method, either method could run faster in practice on given data. More importantly, neither method is very good overall, since the top-down approach may repeatedly establish the same fact, and the bottom-up method may generate many facts that do not contribute in any way to the answer to the query.

A variety of methods that are more efficient than either top-down or bottom up are known. The algorithms of McKay and Shapiro [1981], Pereira and Warren [1983], Lozinski [1984], and Van Gelder [1985] are "lazy evaluators" that do not attempt to establish a fact until some need for that fact has been found. Pereira and Warren call their method *Earley deduction*, and it is actually a method for full Prolog. Porter [85] has shown that Early deduction is convergent and complete for Datalog. These algorithms are all dynamic, in the sense that they make decisions regarding relevance at run time.

There is also an algorithm of Henschen and Naqvi [1984], which is run at compile time to generate an evaluator for a set of rules. The class of problems for which that algorithm is efficient probably does not extend beyond the linear rules, but it will produce a fairly efficient evaluator for the same-generation problem.

The methods we shall discuss are compile-time algorithms to transform the rules into equivalent rules that can be implemented more efficiently bottom-up. Our first method, called "magic sets," is an attempt to perform at compile time the optimization performed by the first group of algorithms at run time. We cannot compare our performance with the run-time algorithms directly, although for linear rules we can see that our algorithm mimics the others. Finally, we give two more ("counting" and "reverse counting") methods that are less generally applicable, but when they are useful, as in the same-generation problem, they can be orders of magnitude more efficient than any of the methods mentioned above, including the "magic set" method.

II. Magic Sets

A desirable property of an algorithm for solving the sg rules is that it only discovers "relevant" sg facts. Intuitively, a fact is "relevant" if it might, depending on the database, be essential to the establishment of a fact that is in the answer. We're not sure that this definition holds water, because there might be a very slow and stupid algorithm that rarely consults the database, and therefore may generate lots of facts that are in principle relevant because of our ignorance of what is really in the database. Another problem is that we are operating in the "all answers" world. We may have to do some work to determine that certain proof trees are invalid for the current database, so we are convinced that we have not overlooked an answer. We do not know whether an evaluation scheme can be complete without generating some facts that are irrelevant to all answers.

Notwithstanding these problems, the diagram of Fig. 1 suggests a plausible definition of the relevant sg facts for query $sg(a, W)$. First, we see the *cone* of a, which is all ancestors of a. We must then generate all facts $sg(b, c)$ such that b is in the cone; c may or may not be in the cone. We may generate all such pairs if we start with the pair $sg(b, b)$ for every b in the cone, and we apply rule r_2 bottom up. However, as we do so, we need not generate facts like $sg(d, f)$, where neither d nor f are in the cone, because such facts could never lead to a fact about a (or else d would be an ancestor of a, and therefore in the cone). On the other hand, given $sg(b, c)$, we do want to establish $sg(e, f)$, if $par(e, b)$ and $par(f, c)$.†

One way to restrict our effort is to define the "magic set" for the first argument of the predicate sg. We do so by determining what values will ever appear as the first argument of sg during top-down evaluation of the goal $sg(a, W)$. Starting with the fact that a can be a value for the first argument, we may observe rules r_1 and r_2 and determine that the only way a value can appear as the first argument of sg is to be either a or the parent of a value that appears there. If we encounter a goal $sg(b, W)$, where b is an

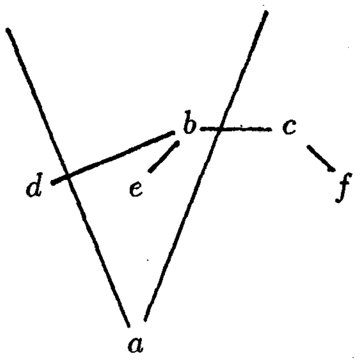


Fig. 1. Efficient computation of $sg(a, W)$.

ancestor of a, and apply r_2, we will get the new goals

$$par(b, X1),\ par(W, Y1),\ sg(X1, Y1).$$

Thus, the first argument, $X1$, of the new sg goal is a parent of b and hence an ancestor of a. In rules:

r_3: $magic(a)$.
r_4: $magic(U)$:- $magic(V)$, $par(V, U)$.

Thus, the set of b's satisfying $magic(b)$ is the cone of a in Fig. 1. While this analysis of possible values for the first argument for sg assumed top-down evaluation, we can use those values to filter bottom-up evaluation. Any proof tree used in bottom-up evaluation will be investigated going top-down. We can then rewrite rules r_1 and r_2 to insist that values of the first argument of sg are in the magic set:

r_5: $sg(X, X)$:- $magic(X)$.
r_6: $sg(X, Y)$:- $magic(X)$, $par(X, X1)$, $par(Y, Y1)$, $sg(X1, Y1)$.

We can show that rules r_3, r_4, r_5, and r_6 yield the same answer to the query $sg(a, W)$ as r_1 and r_2. First, since there are no function symbols, and the rules are all Horn clauses, the least fixed point of each set of rules is well defined and can be computed

† **The reader should be aware that other methods to evaluate this query exist, and some of them, such as Henschen and Naqvi's or the counting method to be described, work by computing something besides *sg* facts. Thus, we cannot even ascribe optimality to an algorithm that computes an apparently minimal set of *sg* facts.**

bottom-up. We must show that an answer $W = w_0$ produced in answer to the query $sg(a, W)$ by rules r_1 and r_2 is also produced by the other set. It is an easy induction on the number of times r_2 is applied, that for every fact $sg(b, c)$ inferred in the proof of $sg(a, w_0)$, $magic(b)$ is true by r_3 and r_4. Once established, we observe that the addition of the literals $magic(X)$ in r_5 and r_6 do not eliminate any facts that are needed to prove $sg(a, w_0)$.

Rules r_3 through r_6 were not quite pulled out of a hat, despite our use of the term "magic sets." There is a general algorithm by which we replace a collection of rules by another collection that is normally more efficient. Sometimes, our algorithm decides that no transformation is desirable, and sometimes it makes a transformation to rules that run more slowly on the given data than the originals.

Before the reader imagines that such a state of affairs is not worthy even of contempt, let us note that the problem of selecting the optimal way to evaluate rules is not as trivial as it might seem. Recent work of Bancilhon [1985a,b] tries to develop definitions of "optimal" evaluators, but notes that even defining the term properly is hard, and there are almost no evaluators that meet his definition. In fact, we may observe that the situation in database query optimization is no better off, despite many years during which the problems have been considered, and despite the models being somewhat simpler.

Example 2: It is conventional wisdom incorporated into every optimizer for relational algebra that selections should be done before joins; see Maier [1983] or Ullman [1982]. Thus, the expression

$$\pi_M(\sigma_{E=\text{'Jones'}}(ED \bowtie DM)$$

would normally be evaluated by first selecting the employee-department relation ED for employee = 'Jones', obtaining a set of departments Jones works for, then selecting those departments from the department-manager relation DM and listing all the managers of those departments. However, it would be more efficient to evaluate the expression as written if Jones were in many departments, and the DM relation were empty, or very small. Then, the supposedly inefficient way to evaluate would rapidly discover that there were no tuples in $ED \bowtie DM$, while the efficient method would have to examine the ED relation and retrieve many departments. □

We conclude from Example 2 that it is unreasonable to expect proofs of optimality, or even superiority of one algorithm over another, in all cases. As with folk wisdom such as "push selections ahead of joins," we must reason from what we believe to be typical data to which the rules are applied and implement evaluation algorithms that perform well in what we regard as the typical case.

Having justified somewhat our lack of optimality results, let us explore the idea behind magic sets. The general idea is to construct the rule/goal graph (Ullman [1985]), showing how bound arguments in the query, such as a in our example $sg(a, W)$, propagate through the rules and predicates. Recall from Ullman [1985] that the nodes of the rule/goal graph correspond to the predicate symbols and rules, and they have "adornments" indicating which arguments of predicates are bound and which variables of rules are bound. Arcs lead to a predicate (goal) node from each rule with that predicate on the left side. Arcs lead to a rule node from each of the literals (= goals) on the right side of the rule. In each case the arcs only connect nodes where the adornments agree. There are some subtleties regarding the proper adornments of these nodes. In particular, in this paper we shall take advantage of the *sideways information passing* strategy explored in Sagiv and Ullman [1984].

Briefly, when evaluating a rule, we evaluate each of the goals on its right side, and we may do so in any order. Whatever order we pick, a variable that appears in a goal may be regarded as bound for all subsequent goals, regardless of whether the adornment of the rule node whose relation we are evaluating says the variable is bound or free. The intuitive justification for this assumption is that we may compute the relations for each goal in the chosen order, and join the relations as we proceed. Thus, any variable that has appeared in a prior goal has already had a limited set of possible values determined for it. Suppose we have an algorithm that computes the relation for some goal $g(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$ provided $X_1, \ldots, X_n$ are bound, and the X_i's have all appeared in previous goals. Then we may consider each tuple in the join of the relations for the previous goals, and from each tuple obtain bound values for $X_1, \ldots, X_n$. We apply the hypothetical algorithm to compute the relation for g with this binding of the X_i's, and take the union over all bindings. The result will be those tuples in the relation for goal g that do not "dangle," i.e., those tuples that actually may join with the relations for the other goals.

Example 3: Figure 2 shows the rule/goal graph for r_1 and r_2 when the order of goals in r_2 is the one indicated in r'_2. The adornments of goal nodes indicate by b or f whether the various arguments of the predicate are bound or free. E.g., sg^{bf} means predicate sg with the first argument bound and the second free, as in $sg(a, W)$. For rule nodes, we indicate that variable X is bound by X/b, and that Y is free by Y/f, for example.

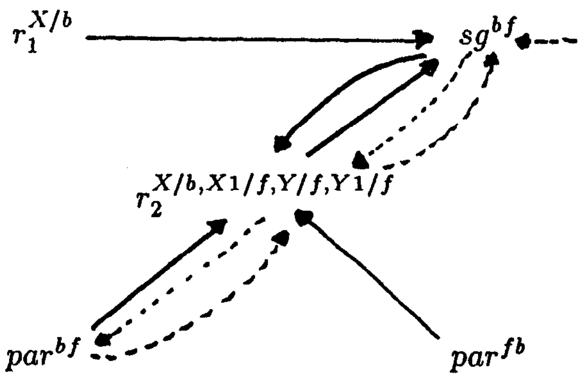


Fig. 2. Rule/goal graph for same-generation problem.

For the moment, ignore the dashed lines and concentrate only on the solid lines, the arcs of the rule/goal graph. To understand the adornments on the predecessors of the r_2 node, note that X is bound. Thus, we may first work on goal node $par(X, X1)$ with the first argument bound, and the result will be a set of values for $X1$ that will in fact be the parents of the individual to which X is bound. Then, for each such value of $X1$ we bind $X1$ to it in the call to goal $sg(X1, Y1)$. Thus, we have another instance of sg^{bf}, and it is the "sideways information passing" that lets us call sg recursively with the first argument bound each time, rather than winding up calling sg with both arguments free, as would be the case with a more naive algorithm. Finally, the call to sg binds $Y1$, and we use the bindings we get in a call to $par(Y, Y1)$ with the second argument bound; this call corresponds to the goal node par^{fb}, which is another predecessor of the r_2 node. □

If we examine the dashed lines in Fig. 2, we see a trace of what happens to a bound value. The initial bound value a is injected as the first argument of sg, shown by the dashed arrow at the top. This bound value becomes the value of X in r_2, and X in r_2 becomes the first argument of par in the call represented by the node par^{bf} in Fig. 2. That value gets translated by evaluation of the database relation par, i.e., we now have a set of values for the second argument of par.

Those values become bindings for $X1$ in the rule r_2, and $X1$ becomes the first argument of sg. We could continue; the first argument of sg gets translated to a set of second arguments, which become values for $Y1$ and the second argument of par. These are translated to values for the first argument of par, which through variable Y becomes the second argument of sg again. However, an important heuristic for the application of magic sets is that it doesn't pay to translate through a relation that we are in the process of computing, for to do so requires that we solve the whole problem just to construct the magic set, which was supposed to be of help solving the problem.

Focussing on the dashed loop in Fig. 2, we can construct the set of all values that can ever be the bound value of the first argument of sg in a tuple that contributes to the answer. We shall give a formal and general algorithm shortly, but in this case we can easily see what is involved. We know that to begin, a is a possible value. This explains rule r_3, $magic(a)$. We also know that if V is a value that has been found useful, then following it around the loop, we find that all those U for which $par(V,U)$ are also useful; this explains rule r_4, $magic(U)$:- $magic(V)$, $par(V,U)$. We shall now formalize the above intuitive discussion.

Algorithm 1: Modification of rules by magic sets.

INPUT: A portion of a rule/goal graph for a set of rules, one goal node of which is the query node. We assume the rules are Horn clauses and have no function symbols. We also assume the predicates are divided into those that are recursive and those that are not. We assume the nonrecursive predicates are not heads of rules, i.e., they are database relations.†

OUTPUT: A modification of the rules (perhaps no change) that is designed to permit efficient evaluation of the rules bottom-up.

METHOD: First, note that since the input is a rule/goal graph, we may assume all decisions regarding the order of literals for sideways information passing have been made; the effectiveness of magic sets depends on the choices made. For each goal node g whose predicate is recursive, we construct a relation $magic_g$, which is the set of tuples of values (for those arguments that the adornment of g says are bound) that can actually contribute to the answer to the query. Observe that we are going to have several magic sets for a predicate that appears in several goal nodes.

Let us assume temporarily that the rules are all linear recursive, and write a typical rule as

$$r\text{: } p \text{ :- } \mathcal{L},\ q,\ \mathcal{R}$$

Here, q is the recursive predicate in the body, $\mathcal{L}$ is the set of literals that are evaluated before q (for the purposes of sideways information passing), and $\mathcal{R}$ the set of literals evaluated later.

Let g be a goal node corresponding to

† In practice, these predicates could be defined by rules, but if so, they do not depend on the recursive predicates of the set of rules in question. This assumption corresponds to the query implementation strategy in which we divide the predicates into strong components, and work on the strong components in the natural "bottom-up" order.

predicate p with some adornment. Then g has a predecessor rule node corresponding to rule r, and that rule node has a predecessor h corresponding to the predicate q. Moreover, the adornment on h indicates an argument of q is bound exactly when the corresponding variable in r is bound. Such a binding comes from a bound argument of p (according to the adornment of g), or from the literals of $\mathcal{L}$.

Suppose that we have discovered that to answer the query we need to evaluate the relation for goal node g with tuple of bound values $\mathbf{b}$, i.e., $\mathbf{b}$ is in $magic_g$. Let $\mathbf{Y}$ be the list of arguments in r that appear in arguments of p that g says are bound (so $\mathbf{b}$ is a tuple of bindings for $\mathbf{Y}$). Let $\mathbf{X}$ be the list of variables in r that appear in arguments of q that h says are bound. Then we may conclude that tuple $\mathbf{c}$ of bindings for $\mathbf{X}$ is in $magic_h$ if the bindings in $\mathbf{c}$ are consistent with the bindings for $\mathbf{b}$, plus some set of legal bindings for the variables in literals of $\mathcal{L}$. Formally, this condition can be expressed:

$$magic_h(\mathbf{X}) \text{ :- } \mathcal{L},\ magic_g(\mathbf{Y}).$$

Notice that p and q have switched sides in the magic set rule, and the literals in $\mathcal{R}$ have disappeared. For example, if r is

$$p(R,S,T) \text{ :- } d(R,U),\ e(U,V),\ q(S,V,W),\ f(W,T)$$

g is p^{bbf} and h is q^{bbf}, then the new rule is

$$magic_h(S,V) \text{ :- } d(R,U),\ e(U,V),\ magic_g(R,S).$$

We create such a magic set rule for each rule that has a recursive predicate in its body, and for each pair of goal nodes g and h related to the rule as above. We complete the magic set rules with

$$magic_g(\mathbf{a}).$$

where g is the goal node corresponding to the query, and $\mathbf{a}$ is the tuple of constants provided by the query.

Next, we rewrite the original rules, as follows. First, we replace each recursive predicate by new predicates corresponding to its nodes in the rule/goal graph, i.e., we distinguish versions of a predicate by the possible sets of bound arguments with which it may be called. Second, we replace a rule such as p :- $\mathcal{L}$, q, $\mathcal{R}$, above, by

1. Replacing p and q by all possible pairs of goal nodes g and h, related to each other and to p and q as above, and
2. Adding literal $magic_g(\mathbf{Y})$ to the body; $\mathbf{Y}$ is the list of variables appearing in the argument positions of p that g asserts are bound.

That is, the rule becomes:

$$g \text{ :- } magic_g(\mathbf{Y}),\ \mathcal{L},\ h,\ \mathcal{R}.$$

If the rule has no recursive predicate on the right, we still perform step (2), i.e., insertion of the magic set term on the right. Note that the "magic" rules do not refer to the recursive predicates. Thus, in bottom-up evaluation of the combined set of rules, we may choose to run the magic rules to completion first.

The last point to be considered is what happens when there is more than one recursive literal in a rule body. If none of these appear in $\mathcal{L}$, no change needs to be made. If $\mathcal{L}$ contains recursive literals, often we cannot do anything, and the algorithm simply fails to rewrite the given rules. The case where we can proceed is when none of the recursive literals in $\mathcal{L}$ is necessary to bind an argument of q. In that case, we can transfer the recursive literals from $\mathcal{L}$ to $\mathcal{R}$, and proceed as above. We shall see in Example 5 a case where a rule with two recursive literals is not fatal. □

Example 4: Let us consider again the same-generation problem, but with the rules rewritten so the recursive call to sg occurs with its arguments in the reverse order. The rules are:

r_1: $sg(X,X)$.
r_7: $sg(X,Y)$:- $par(X,X1)$, $par(Y,Y1)$, $sg(Y1,X1)$.

The query is again $sg(a,W)$. Figure 3 shows the relevant portion of the rule/goal graph.

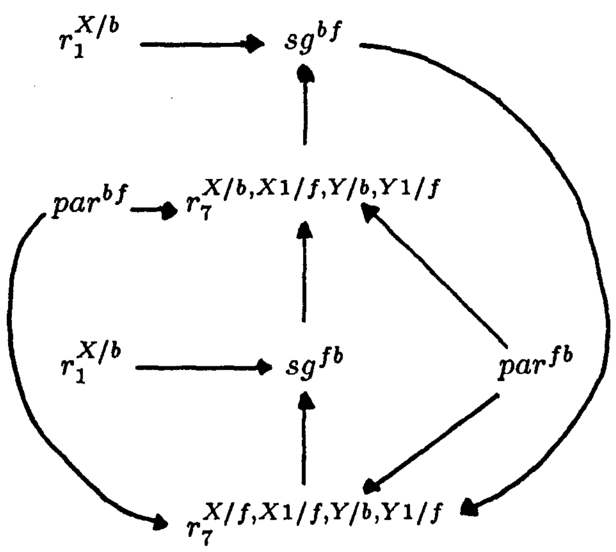


Fig. 3. Rule/goal graph for modified same-generation problem.

Although there is only one recursive predicate, sg, there are two goal nodes for that predicate, one with the first argument bound and one for the second argument bound. Let us use $magic_1$ for node sg^{bf} and $magic_2$ for sg^{fb}.

For the case in Algorithm 1 where g is sg^{bf} and h is sg^{fb}, the order of literals in r_7 used for sideways information passing is $par(X,X1)$, $sg(Y1,X1)$, $par(Y,Y1)$. Thus, $\mathcal{L}$ is $par(X,X1)$. The magic set rule is therefore:

$magic_2(X1)$:- $par(X,X1)$, $magic_1(X)$.

Note that X is the argument of $magic_1$ because X is in the bound position of sg on the left of r_7, and $X1$ is the argument of $magic_2$ because that is in the bound position of sg on the right. The rules for sg are modified into:

$sg^{bf}(X,X)$:- $magic_1(X)$.
$sg^{bf}(X,Y)$:- $magic_1(X)$, $par(X,X1)$ $par(Y,Y1)$, $sg^{fb}(Y1,X1)$.

Now, let g be sg^{fb} and h be sg^{bf}. The order of the literals of r_7 that provides the desired sideways information passing is

$par(Y,Y1)$, $sg(Y1,X1)$, $par(X,X1)$

This order yields the magic set rule:

$magic_1(Y1)$:- $par(Y,Y1)$, $magic_2(Y)$.

and the modified sg rules:

$sg^{fb}(X,X)$:- $magic_2(X)$.
$sg^{fb}(X,Y)$:- $magic_2(X)$, $par(X,X1)$, $par(Y,Y1)$, $sg^{bf}(Y1,X1)$.

Finally, the query node is sg^{bf}, with the first argument bound to a. Thus we have the basis magic set rule:

$magic_1(a)$.

the seven rules, with some renaming of variables, are as follows.

$magic_1(a)$.
$magic_1(U)$:- $par(V,U)$, $magic_2(V)$.
$magic_2(U)$:- $par(V,U)$, $magic_1(V)$.

$sg^{bf}(X,X)$:- $magic_1(X)$.
$sg^{bf}(X,Y)$:- $magic_1(X)$, $par(X,X1)$, $par(Y,Y1)$, $sg^{fb}(Y1,X1)$.

$sg^{fb}(X,X)$:- $magic_2(X)$.
$sg^{fb}(X,Y)$:- $magic_2(X)$, $par(X,X1)$, $par(Y,Y1)$, $sg^{bf}(Y1,X1)$.

Notice that the two magic sets are almost identical, but the generations are counted modulo 2 starting with individual a, so they may not be the same set. A similar comment applies to the two versions of the *sg* predicate. □

Example 5: The rule

$$p(X,Y) \text{ :- } a(X,Z),\ b(X,W),\ p(Z,U),\ p(W,V),\ c(U,V,Y).$$

is an example where a nontrivial magic set may exist, even though the rule is nonlinear. Suppose that the query node is p^{bf}, and that this is the only node for p in the rule/goal graph. Also assume that the literals are to be evaluated in the order shown. For the literal $p(Z,U)$ we let $\mathcal{L} = a(X,Z),\ b(X,W)$, so we can write

$$magic_p(Z) \text{ :- } a(X,Z),\ b(X,W),\ magic_p(X).$$

Now consider the second literal for p, that is, $p(W,V)$. It is preceded by another literal for p, but that literal, $p(Z,U)$, has no variable in common with $p(W,V)$. Thus, with respect to $p(W,V)$ we can exclude $p(Z,U)$ from $\mathcal{L}$, so $\mathcal{L}$ again equals $a(X,Z),\ b(X,W)$. The magic set rule derived from the second occurrence of p is thus:

$$magic_p(W) \text{ :- } a(X,Z),\ b(X,W),\ magic_p(X).$$

□

Theorem 1: The modified rules produced by Algorithm 1 compute the same answer to the query as the original rules. □

III. The Counting Method

If we review Fig. 1, we see that there is potential redundancy in our intent to calculate all pairs (b,c) such that b is in the cone, and $sg(b,c)$. For example, a may have 100 ancestors $b_1,\ldots,b_{100}$ at the 10th generation, and we would therefore discover the following 100 facts: $sg(b_1,c),\ldots,sg(b_{100},c)$. But any one of these facts is sufficient to deduce the fact that all 10th-generation descendants of c are same-generation cousins of a.

The matter isn't all that clearcut, because b may be an ancestor of a at several different generations. If that is the case, the fact $sg(b,c)$ serves several purposes at different times, and replacing it by statements that c was a cousin of some ancestor of a at these various generations could increase our work rather than decrease it.

The next section offers some comparisons among various strategies, but the transformation we propose for the same-generation rules has intuitive appeal, and in fact we believe it is closely related to the Henschen-Naqvi algorithm. We shall not attempt to generalize the idea as we did for magic sets, but the following example should make the potential idea clear.

While the magic set method, applied to rules r_1 and r_2 computes the cone of a, as depicted in Fig. 1, we could, with a little more effort, compute for each member of the magic set the levels at which it appears, that is, the number of generations removed from a it is. Recall that an individual can appear on several levels. The rules to compute this information are:

$$ancestor(a,0).$$
$$ancestor(X,I) \text{ :- } par(Y,X),\ ancestor(Y,J),\ I \text{ is } J+1.$$

Now, we can compute a new predicate $cousin(X,I)$, meaning that the Ith-generation descendants of X are cousins of a at the same generation. The rules are:

$$cousin(X,I) \text{ :- } ancestor(X,I).$$
$$cousin(X,I) \text{ :- } par(X,Y),\ cousin(Y,J),\ I \text{ is } J-1,\ I \geq 0.$$
$$sg(a,X) \text{ :- } cousin(X,0).$$

We call the idea that led to the above rules the *counting method*, and note that this method can answer the query $sg(a,W)$

correctly, but only if the parenthood relation is acyclic. If there are cycles, we never get finished computing the ancestor relation. If there are n individuals in the database, then one can show that I in the rules for *ancestor* and *cousin* can be limited to n^2, but it is not always possible to consider only the first n generations.

IV. The Reverse Counting Method

Yet another method, called *reverse counting*, starts off by computing the magic set, forgetting about the levels at which each individual appears. Then, we consider each element b of the magic set, and we compute, for all i, the set of descendants of b at the ith generation. For those values of i for which the ith descendants of b contain a (recall a is the individual whose cousins we are computing), we add all descendants at level i to the set $sg(a, X)$. That is, after computing the magic set for a, we create a predicate *b-descendant*(X, I) for each b such that *magic*(b), with rules:

$$\begin{array}{l} b\text{-}descendant(b, 0). \\ b\text{-}descendant(X, I) \text{ :- } \\ \qquad b\text{-}descendant(Y, J), \\ \qquad par(X, Y),\ I \text{ is } J + 1. \end{array}$$

Then we contribute to the solution by:

$$\begin{array}{l} sg(a, X) \text{ :- } b\text{-}descendant(a, I), \\ \qquad b\text{-}descendant(X, I). \end{array}$$

Remember that there is one set of the above rules for each value of b.

Intuitively, the reason the reverse counting method can sometimes be more efficient than other methods is that by focussing on one value of b at a time, it can compute all the relevant sg pairs not as pairs, but as a single set. Put another way, if *b-descendant*(c, i) and *b-descendant*(d, i), then $sg(c, d)$ follows. However, if the set of ith descendants of all magic set members, or even all magic set members at a given height were computed in one group, the membership of two elements c and d in the set of desdendants at level i would not be sufficient to guarantee $sg(c, d)$. Note that similarly to the counting method, reverse counting loops forever if the parenthood relation is cyclic.

V. Comparisons Among Methods

First, let us note that the magic set method, like other compiled methods, cannot always compete with the dynamic methods like Those of McKay-Shapiro, Pereira & Warren, Lozinski, or Van Gelder. A simple example is the transitive closure rules

$$\begin{array}{l} t(X, Y) \text{ :- } e(X, Y). \\ t(X, Y) \text{ :- } t(X, Z),\ t(Z, Y). \end{array}$$

The magic set, counting, reverse counting, and Henschen-Naqvi methods are useless given the query $t(a, W)$, because to find the possible values of Z, which all belong in the magic set, we have to compute $t(a, Z)$, which is equivalent to the query. Yet the dynamic methods will correctly perform a breadth-first search from a. However, in some cases the magic set method is better than dynamic methods. For example, Lozinski's method fails to compute only relevant facts when given the rules of Example 4.

When we compare compiled methods we can make some interesting and concrete observations. In what follows, we use a pair of rules similar to the same-generation rules, but ones that allow us better control over the performance of the various algorithms.

$$\begin{array}{l} r_8\text{: } p(X, Y) \text{ :- } q(X, Y). \\ r_9\text{: } p(X, Y) \text{ :- } r(X, X1),\ p(X1, Y1), \\ \qquad s(Y1, Y). \end{array}$$

In what follows, it helps to see r as "going up"; if $r(a, b)$, then we place b above

a and draw an arc $a \rightarrow b$. We see q as going sideways; if $q(a,b)$ we place b to the right of a and draw an arc $a \rightarrow b$. Finally, s represents down arcs, and when $s(a,b)$ is true, we place b below a and have and arc $a \rightarrow b$. In the following examples we assume that whenever a selection has to be performed on a database relation, there is an index that let us perform that selection in time proportional to the size of the result (in particular, in constant time if the result has a single tuple).

Example 6: This example shows that the magic set method can be worse than either the Henschen-Naqvi, counting, or reverse counting methods, even on linear rules. Let q, r, and s be defined by the graph of Fig. 4. That is, the database facts are

1. $r(a,b_i)$ and $r(b_i,c)$ for $1 \leq i \leq n$.
2. $q(c,d)$.
3. $s(d,e_i)$ and $s(e_i,f)$ for $1 \leq i \leq n$.

The magic set for the query $p(a,W)$ is $\{a,c,b_1,\ldots,b_n\}$. We then establish all facts $p(b_i,e_j)$, taking $\Omega(n^2)$ time since there are n^2 such facts.

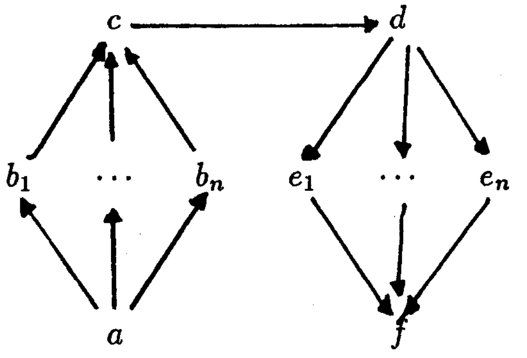


Fig. 4. Example where the Henschen-Naqvi method is most efficient.

On the other hand, Henschen & Naqvi's algorithm will establish in $O(n)$ time the fact that $\{b_1,\ldots,b_n\}$ are the ancestors of a at generation 1, c is the only ancestor at generation 2, and find the only answer, $W = f$, in another $O(n)$ time by walking down the right side of Fig. 4. The counting method behaves in essentially the same way.

The reverse counting method computes the magic set in $O(n)$ time, and then for the member c, it finds in $O(n)$ time that a is two levels below c while f is the same number of levels below d and, hence, gets the answer $W = f$. In $O(n)$ time it determines that c is the only member of the magic set that can produce an answer (since only c appears in q) and, so, $W = f$ is the only answer. □

Example 7: Now we take up a family of databases for the same rules as Example 6, for which magic sets is more efficient than the other algorithms. The database consists of:

1. $r(a_i,a_{i+1})$ for $1 \leq i < n$.
2. $r(a_1,a_i)$ for $3 \leq i \leq n$.
3. $q(a_n,b_n)$.
4. $s(b_i,b_{i-1})$ for $2 \leq i \leq n$.

The case $n = 5$ is shown in Fig. 5. The dashed lines illustrate the p facts that we shall eventually infer.

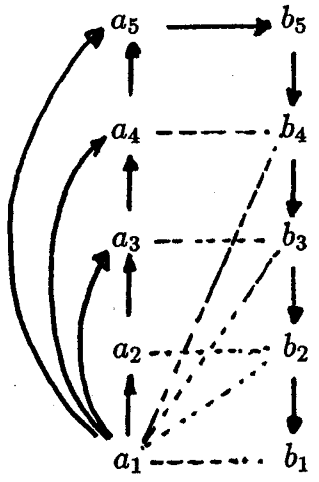


Fig. 5. Example where magic sets is most efficient.

The magic set for query $p(a_1, W)$ consists of all the a_i's, and it is computed in $O(n)$ time. Once a_n is in the magic set, we can use r_8 to establish $p(a_n, b_n)$. Then on the next pass we establish $p(a_{n-1}, b_{n-1})$ and $p(a_1, b_{n-1})$. On the next pass, we find both $p(a_{n-2}, b_{n-2})$ and $p(a_1, b_{n-2})$, and so on. The total work is seen to be $O(n)$.

In comparison, Henschen-Naqvi and the counting method both take $\Omega(n^2)$ time just to establish the ancestry of a_1. That is, the set of first-generation ancestors of a_1 is

$$\{a_2, \ldots, a_n\}$$

the set of second-generation ancestors of a_1 is

$$\{a_3, \ldots, a_n\}$$

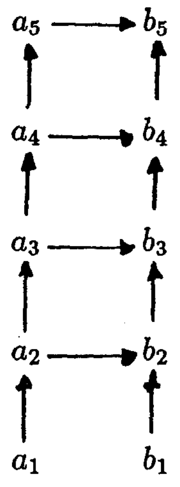
and so on.

Interestingly, the reverse counting method takes only $O(n)$ time. After computing the magic set, this method finds in $O(n)$ time that the i^{th} level below a_n has two elements, namely, a_1 and a_{n-i}. The i^{th} level below b_n consists exactly of b_{n-i} and, so, the answers $b_{n-1}, \ldots, b_1$ are found in $O(n)$ time. In another $O(n)$ time the reverse counting method determines that a_n is the only member of the magic set that produces some answers. □

Example 8: A similar example, shown in Fig. 6 for the case $n = 5$, demonstrates that counting can be much better than either the Henschen-Naqvi or the reverse counting methods. Here, the database is:

1. $r(a_i, a_{i+1})$ for $1 \leq i < n$.
2. $q(a_i, b_i)$ for $2 \leq i \leq n$.
3. $s(b_i, b_{i-1})$ for $2 \leq i \leq n$.

Clearly, the counting method establishes the facts that a_i is an $(i-1)$st ancestor of a_1 in $O(n)$ time, and can also establish all the "cousin" facts, namely that b_i is a cousin of a_1 $i-1$ generations removed, in $O(n)$ time.



$$\begin{tikzcd}
a_5 \arrow[r] & b_5 \\
a_4 \arrow[r] \arrow[u] & b_4 \arrow[u] \\
a_3 \arrow[r] \arrow[u] & b_3 \arrow[u] \\
a_2 \arrow[r] \arrow[u] & b_2 \arrow[u] \\
a_1 \arrow[u] & b_1 \arrow[u]
\end{tikzcd}$$

Fig. 6. Example where counting is most efficient.

Magic sets similarly works in time $O(n)$. However, Henschen-Naqvi, given that b_i is a cousin i generations removed, runs down the chain $b_{i-1}, \ldots, b_1$, and it does so for every i. Thus, Henschen-Naqvi must take $\Omega(n^2)$ time. The reverse counting method behaves in essentially the same way.

Actually, we may be somewhat unfair to Henschen-Naqvi. If they retained the facts they had established "coming down" on earlier passes and compare new facts with them, then they could detect that the same chain was being followed repeatedly, and reduce their running time on this example to $O(n)$. In a sense, the counting strategy can be seen as an implementation of Henschen-Naqvi that energetically tries to avoid repeating an inference. But neither algorithm deals specifically with the case that the relation r has cycles, and it is not clear that we want to run either in cases where we could not guarantee r was acyclic, as we might expect the parenthood relation of our earlier examples to be. □

The performance of the four algorithms on the three families of databases described in this section is summarized in the following table.

Example	6	7	8
Henschen-Naqvi	n	n^2	n^2
Magic Sets	n^2	n	n
Counting	n	n^2	n
Reverse Counting	n	n	n^2

VI. Open Problems

1. Can we modify the magic set idea so it applies to sets of rules that are nonlinear in a nontrivial way, such as the transitive closure rules mentioned at the beginning of Section V?
2. How may the counting method and the reverse counting methods be generalized from the same-generation example to arbitrary linear rules, in analogy with the way Algorithm 1 generalizes Example 3? What is the relationship between counting and Henschen and Naqvi's algorithm?
3. Can one characterize the rules and databases for which one algorithm is superior to another?
4. Is there an algorithm that provably dominates the algorithms defined or referenced in this paper on arbitrary rules and databases? The reader should review Example 2, because it points out the fact that we have to be careful with our definitions in order to rule out pathological databases. The "right" definition of the performance of an algorithm for evaluating queries in logical databases is itself an open problem.

References

F. Bancilhon [1985a]. "Performance of rule based systems," unpublished manuscript, MCC, Austin, TX.

F. Bancilhon [1985b]. "Naive evaluation of recursively defined relations," unpublished manuscript, MCC, Austin, TX.

D. Brough and A. Walker [1984]. "Some practical properties of logic programming interpreters," *Proc. Japan FGCS84 Conf.*, November 1984, pp. 149-156.

H. Gallaire and J. Minker (eds.) [1978]. *Logic and Databases*, Plenum, New York.

E. L. Lozinski [1984]. "Inference by generating and structuring of deductive databases," Report 84-11, Dept. of CS, Hebrew Univ.

D. Maier [1983]. *The Theory of Relational Databases*, Computer Science Press, Rockville, Md.

D. Maier and D. S. Warren [1985]. *Introduction to Logic Programming*, unpublished memorandum, Oregon Graduate Center.

D. McKay and S. Shapiro [1981]. "Using active connection graphs for reasoning with recursive rules," *Proc. 7th IJCAI*, pp. 368-374.

F. C. N. Pereira and D. H. D. Warren [1983]. "Parsing as deductions," *Proc. 21st Meeting of the ACL*, June 1983.

H. H. Porter, III [1985]. "Earley deductions," unpublished manuscript, Oregon Graduate Center.

Y. Sagiv and J. D. Ullman [1984]. "Complexity of a top-down capture rule," STAN–CS–84–1009, Dept. of CS, Stanford Univ.

J. D. Ullman [1982]. *Principles of Database Systems*, Computer Science Press, Rockville, Md.

J. D. Ullman [1985]. "Implementation of logical query languages for databases," *ACM Trans. on database Systems* **10:3**, pp. 289–

321.

A. Van Gelder [1985]. "A Message Passing Framework for Logical Query Evaluation," Tech. Report, Dept. of CS, Stanford Univ.