PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

A QUERY INDEPENDENT METHOD FOR MAGIC SET COMPUTATION ON STRATIFIED DATABASES

Isaac Balbin

Department of Computer Science
Royal Melbourne Institute of Technology,
GPO Box 2476 V, Melbourne, Australia
isaac%goanna.oz@uunet.uu.net

Krishnamurthy Meenakshi
Kotagiri Ramamohanarao
Department of Computer Science,
University of Melbourne,
Parkville 3052, Australia.
rao%munnari.oz@uunet.uu.net

ABSTRACT

A semantics for stratified and allowed databases is presented in (Apt et al., 1988). Based on this semantics, a method to compute the answers to a query using magic sets is given in (Balbin et al., 1987). The method is based upon a labeling algorithm (the BPR algorithm) which separates the context of predicates in a rule. Labeling ensures that stratification is preserved when the magic rules for positive literals are constructed. We present a new efficient labeling algorithm that has the virtues of generating a polynomial number of new predicates (in the worst case), and of requiring no re-labeling of the database because it is independent of subsequent user queries.

1 INTRODUCTION

Based on a query, magic set algorithms (Bancilhon et al., 1986; Beeri and Ramakrishnan, 1987; Rohmer et al., 1986) perform a compile-time transformation of a database into an equivalent form that enables a standard or enhanced (Bancilhon, 1985; Balbin and Ramamohanarao, 1987) bottom-up computation to focus on *relevant* tuples. In this paper, we restrict ourselves to *magic set* transformations on function-free, stratified (Chandra and Harel, 1985) and allowed (Clark, 1978) databases. Magic set algorithms for these databases are based on allowed *sideways information passing strategies* (*sips*) (Beeri et al., 1987; Balbin et al., 1987).

The paper is organised as follows. Notation and preliminary definitions are presented in the rest of this section. In section 2 we motivate the reason for labeling. In section 3 we identify two deficiencies with a previous labeling algorithm. Firstly, it may require the re-labeling of the database when the user issues a new query. Unlike the case for a database without negative literals, the program will often have to be re-compiled as a result of new queries. Secondly, the number of new predicates generated may be exponential in the number of predicates in the original database. We present an efficient labeling algorithm that overcomes these difficulties. It has the virtues of generating a polynomial number of new predicates (in the worst case), and of requiring no re-labeling of the database because it is independent of subsequent queries.

We denote variables by identifiers beginning with an upper case letter and constants by identifiers starting with a lower case letter. In the absence of function symbols, a *term* is either a constant or a variable. Identifiers starting with lower case letters are used for *predicate* or *relation* names. An *atom* is of the form $p(t_1, t_2, \ldots, t_n)$, $n \geq 0$, where p is a predicate name and $t_1, t_2, \ldots, t_n$ are terms. A *literal* is either an atom, or an atom preceded by the negation sign, $\neg$. A *rule* is a statement of the form $p_0 \leftarrow p_1, p_2, \ldots, p_m$, $m \geq 0$, where p_0 is an atom and $p_1, p_2, \ldots, p_m$ are distinct literals. The atom p_0 is called the *head*, the conjunction $p_1, p_2, \ldots, p_m$ is called the *body*, and each p_i is a *body literal*. Without loss of generality, a *query* q where q is an atom, is a statement of the form $\leftarrow q$. An atom $p(t_1, t_2, \ldots, t_n)$ is ground when all its terms, $t_1, t_2, \ldots, t_n$, are ground; $\langle t_1, t_2, \ldots, t_n \rangle$ is known as a *tuple* of p. A *fact* is a ground rule with no body ($m = 0$). A *base predicate* is defined solely by facts. A *derived predicate* is not defined by any facts. A *derived (base) literal* is one whose predicate is derived (base). A rule whose head predicate is derived, is known as a *derived rule*. The set of derived rules is also known as the *program*. Without loss of generality, a *deductive database* (or simply database) is a finite set of rules consisting of a program **P** and a set of facts **F**.

For a database D, we construct a dependency graph $\mathcal{G}$ (Apt et al., 1988) representing a *refers to* relationship between the predicates. There is a node for each predicate and a directed arc from node a to node b if a is a body literal in a rule whose head is b. When a is a negative literal the arc is said to be a *negative arc*; otherwise it is a *positive arc*. A predicate a *depends on* a predicate b if there is a path of length greater than or equal to one from b to a (*depends on* is the transitive closure of the *refers to* relation). We denote the relation a *depends on* b by $a \overset{*}{\leftarrow} b$. A predicate a is *recursive* if $a \overset{*}{\leftarrow} a$.

A database **D** is *stratified* if and only if there does not exist a negative cycle in the dependency graph for **D**. A *negative cycle* is a cycle where at least one arc in the cycle is negative. A partitioning of **D**'s rules into the sets $\mathbf{D}_0, \ldots, \mathbf{D}_n$ is a *stratification* of **D** if the following conditions hold for $i = 0, \ldots, n$: (1) if a predicate p occurs in $\mathbf{D}_i$ as a positive body literal, then its definition is contained in $\cup_{j \leq i} \mathbf{D}_j$; (2) if a predicate p occurs in $\mathbf{D}_i$ as a negative body literal, then its definition is contained in $\cup_{j < i} \mathbf{D}_j$. The *definition* of a predicate p is the subset of **D** consisting of all rules containing p in the head. Each $\mathbf{D}_i$ is called a *stratum*, and each i is called a *level*. **D** has a stratification if and only if it is stratified (Apt et al., 1988).

A convenient way to analyse the transformation by the magic set algorithm on **D**, is to abstract $\mathcal{G}$. A strongly connected component of $\mathcal{G}$ is a subgraph $\mathcal{G}_s$ such that there is a path of length ≥ 0 between each pair of nodes in $\mathcal{G}_s$. The *condensation* $\mathcal{G}^*$ is a directed graph derived from the maximal strongly connected components (MSCC's) of $\mathcal{G}$. Each node in $\mathcal{G}^*$ corresponds to a MSCC in $\mathcal{G}$.

A stratification $\mathbf{D}_0, \ldots, \mathbf{D}_n$ is *maximal* if (1) for every stratum $\mathbf{D}_i$, $1 \leq i \leq n$, either $\mathbf{D}_i$ contains exactly the rules defining a derived predicate p if p is not recursive; or $\mathbf{D}_i$ contains exactly the rules defining p and (any) other predicates in the same MSCC as p if p is recursive; and (2) $\mathbf{D}_0$ contains all the base predicates. When we assume a maximal stratification, the derived predicates defined in each stratum correspond to the predicates comprising a MSCC .

2 LABELING

2.1 Review of Magic Sets

For the sake of completeness, we briefly outline the concept of magic sets. The reader should refer to (Beeri and Ramakrishnan, 1987; Beeri et al., 1987; Rohmer et al., 1986; Balbin et al., 1987) for various details.

The virtue of the magic set approach to query evaluation is that it permits an efficient bottom-up computation for all types of queries. A pure bottom-up computation is efficient provided that atoms, such as the query atom, do not contain ground terms (Bancilhon and Ramakrishnan, 1988; Han and Lu, 1986).

Consider a program for the predicate prone, which identifies people who are prone to getting a certain disease. The base predicate `parent(X,Y)` is true if `Y` is the parent of `X`, and the base predicate `infected(X)` is true if `X` has been tested and found to be `infected`.

Example 1

```
prone(X) ← infected(X)
prone(X) ← parent(X,Y), prone(Y)
```

When we query ← `prone(randy)`, a bottom-up computation retrieves *all* the people who are prone, only then checking to see whether `randy` is one of these.

Magic set algorithms (Beeri and Ramakrishnan, 1987) transform this program into an equivalent one (with respect to the query) based on sideways information passing strategies (sips) that are depicted by labeled bipartite graphs. Briefly, sips state what bound values are passed between one literal and another inside a rule. An arc in the graph corresponding to the second rule might be

$$\{\texttt{prone(X)}, \texttt{parent(X,Y)}\} \rightarrow_{\texttt{Y}} \texttt{prone(Y)}.$$

This specifies that the body literal `prone` can expect bound values for `Y` via the tail of the arc, `prone(X)` and `parent(X,Y)`. We omit explicit sips for simplicity of exposition, and assume a default sip, where the tail of the sip arc for each body literal q includes all literals to the left of q in the rule (including the head).

In the context of bottom-up computation, magic set algorithms implement the desired information passing by transforming the program and adding a magic fact.

Example 1 (*continued*)

```
prone(X) ← magic_prone(X), infected(X)
prone(X) ← magic_prone(X), parent(X,Y), prone(Y)
magic_prone(Y) ← magic_prone(X), parent(X,Y)
magic_prone(randy)
```

The search is now directed according to *relevant facts*, that is, the ancestors of `randy`. These relevant facts, otherwise known as the *magic set*, are the tuples satisfying `magic_prone`. (Note that common sub-expression elimination is performed by employing supplementary magic sets (Sacca and Zaniolo, 1987))

2.2 Magic Sets and Negation

When the generalised magic set algorithm is applied to a stratified and allowed database the transformed database is not necessarily stratified. A solution to this problem (Balbin et al., 1987) uses the BPR labeling algorithm to partition the magic sets according to the context in which they are constructed. For databases that do not include negative body literals, we construct a single magic set for each positive derived predicate. (In the example above, prone was the only derived predicate and so `magic_prone` was the only magic set created).

For databases with negative body literals, the context in which the magic rule corresponding to a predicate is constructed, is distinguished by a label. We illustrate this idea by an example. Consider example (1) with two additional rules. The first rule states that it is (unfortunately) necessary to `isolate` person `X`, if `X` is a `male` who is prone to the disease and has had relations

with another male partner who is also prone. The second rule states that an antidote for the latent germ is available for X if X is a female who is not prone to the disease even though she may have had relations with a partner who was prone.

Example 2

```
prone(X) ← infected(X)
prone(X) ← parent(X, Y), prone(Y)
isolate(X) ←male(X), prone(X), partner(X, Y), male(Y),
             prone(Y)
antidote(X) ←female(X), ¬prone(X), partner(X, Y),
              prone(Y)
```

Consider the query ← isolate(randy). The following magic rules and modified rules are *relevant*. A rule is *relevant* if the query predicate *depends on* the head of the rule.

```
magic_isolate(randy)
magic_prone(Y) ← magic_prone(X), parent(X, Y)
magic_prone(X) ← magic_isolate(X), male(X)
magic_prone(Y) ←magic_isolate(X), male(X), prone(X),
                 partner(X, Y), male(Y)
isolate(X) ←magic_isolate(X), male(X), prone(X),
            partner(X, Y), male(Y), prone(Y)
prone(X) ← magic_prone(X), infected(X)
prone(X) ← magic_prone(X), parent(X, Y), prone(Y)
```

There are three magic rules corresponding to prone. The first is due to the body literal prone in the definition of prone itself. The second and third rules, however, are derived from the body predicates prone in the rule defining isolate. Although there are three magic rules for prone, there is only one magic set constructed for prone; the tuples satisfying magic_prone. In this example, there is no reason to differentiate between the magic rules by constructing separate magic sets for magic_prone.

Now consider the query ← antidote(petra) which evaluates to true if it is beneficial to give petra the antidote.

```
magic_antidote(petra)
magic_prone(Y) ← magic_prone(X), parent(X, Y)
magic_prone(Y) ←magic_antidote(X), female(X),
                 ¬prone(X), partner(X, Y)
magic_prone(X) ← magic_antidote(X), female(X)
antidote(X) ←magic_antidote(X), female(X),
              ¬prone(X), partner(X, Y), prone(Y)
prone(X) ← magic_prone(X), infected(X)
prone(X) ← magic_prone(X), parent(X, Y), prone(Y)
```

The second rule defining magic_prone, which is derived from the positive literal prone(Y) in the rule defining antidote, introduces the negative cycle

$$\text{prone} \overset{*}{\leftarrow} \text{magic\_prone} \overset{*}{\leftarrow} \neg\text{prone}$$

---

until D has no unlabeled positive derived literals **do**
  label derived positive body literals in D
  for each such labeled literal **do**
   based on the original unlabeled rule in D
   construct the defining rules for the literal
  od
  add the new rules to D
od

Figure 1: BPR labeling algorithm

---

into the *refers to* graph. This source of unstratification is a direct consequence of the fact that (1) the modified rules of the transformed database contain a magic literal as the first body literal; and (2) the magic rules for positive body literals have been constructed in the usual way.

The essential difference between the two queries is one of *context*. With the query ← antidote(petra), magic_prone *depends on* the negative literal ¬prone(Y). In order to compute only (and all) ground instances of the query that are in the intended model $M_{\mathbf{D}}$ (Apt et al., 1988), a specific control discipline based on the strata must be exercised. For the query ← isolate(randy), however, the predicate magic_prone is not dependent on a negative literal and its evaluation does not, therefore, require the evaluation of predicates on a strictly lower stratum before it can proceed.

A solution to this problem, using the BPR algorithm (Balbin et al., 1987), effectively separates the context in which prone appears as a body literal in the rule by labeling one of them as prone_1. The rules that define prone_1 are then simply duplicated from prone. Since prone is recursive, in order to maintain the proper separation between labeled and unlabeled predicates every body literal in rules defining prone that is in the same MSCC as prone is labeled in the same way. (In this case, prone is in a MSCC on its own).

An informal presentation of the BPR algorithm is given in figure 1. After applying the algorithm, the relevant program for the antidote query is

Example 2 (*continued*)

```
prone(X) ← infected(X)
prone(X) ← parent(X, Y), prone(Y)
prone_1(X) ← infected(X)
prone_1(X) ← parent(X, Y), prone_1(Y)
antidote(X) ←female(X), ¬prone(X), partner(X, Y),
              prone_1(Y)
```

The relevant modified rules and magic rules are

```
magic_antidote(petra)
magic_prone(Y) ← magic_prone(X), parent(X, Y)
magic_prone(X) ← magic_antidote(X), female(X)
```

```
magic_prone_1(Y) ← magic_prone_1(X), parent(X, Y)
magic_prone_1(Y) ←magic_antidote(X), female(X),
                   ¬prone(X), partner(X, Y)
prone(X) ← magic_prone(X), infected(X)
prone(X) ← magic_prone(X), parent(X, Y), prone(Y)
prone_1(X) ← magic_prone_1(X), infected(X)
prone_1(X) ←magic_prone_1(X), parent(X, Y),
             prone_1(Y)
antidote(X) ←magic_antidote(X), female(X),
              ¬prone(X), partner(X, Y), prone_1(Y).
```

and the database is stratified.

Labeling algorithms ensure that the stratification is preserved when the magic rules for positive body literals are constructed. In general, when the magic rules corresponding to negative literals are included in the transformed database, a slightly modified bottom-up computation on the labeled database is required to preserve the semantics, since the database may be unstratified. For the sake of completeness we informally describe this using an example.

Consider the prone database of example 1. It is now found if a person is injected with a natural antibody that they do not risk contracting the disease. In addition, it is found that if saliva-based contact has occurred between a person who has the antibody and another person, that the latter person is also not in risk (and doesn't require explicit vaccination). For the query ← norisk(sandy) the program can be expressed by

Example 3

```
prone(X) ← infected(X)
prone(X) ← parent(X, Y), prone(Y)
norisk(X) ← antibody(X)
norisk(X) ← ¬prone(X), contact(X, Y), norisk(Y).
```

The relevant transformed rules after labeling are listed below.

```
magic_norisk_1(sandy)
magic_norisk_1(Y) ←magic_norisk_1(X), ¬prone(X),
                   contact(X, Y)
magic_prone_1(Y) ← magic_prone_1(X), parent(X, Y)
magic_prone(X) ← magic_norisk_1(X)
prone(X) ← magic_prone(X), infected(X)
prone(X) ← magic_prone(X), parent(X, Y), prone(Y)
prone_1(X) ← magic_prone_1(X), infected(X)
prone_1(X) ←magic_prone_1(X), parent(X, Y),
             prone_1(Y)
norisk_1(X) ← magic_norisk_1(X), antibody(X)
norisk_1(X) ←magic_norisk_1(X), ¬prone(X),
              contact(X, Y), norisk_1(Y).
```

Note that labeling alone is not *necessarily* a total solution. As proved by proposition 2, the negative cycle

prone$\overset{*}{\leftarrow}$magic_prone$\overset{*}{\leftarrow}$magic_norisk_1$\overset{*}{\leftarrow}$¬prone

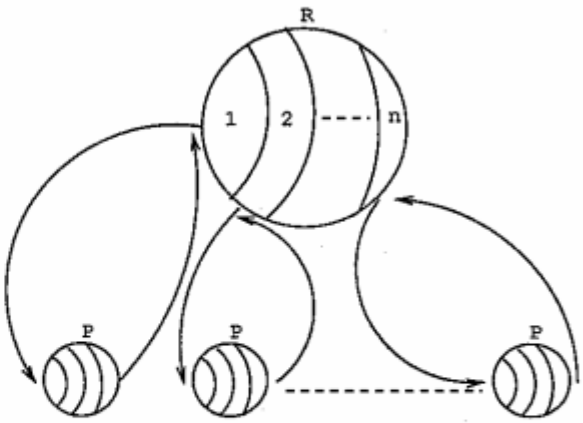

Figure 2: Structured Execution

is introduced by the magic rule constructed for the negative literal ¬prone(X). Rules constructed from negative literals such as

$$\texttt{magic\_prone(X)} \leftarrow \texttt{magic\_norisk\_1(X)} \qquad (1)$$

are not included in the program in the usual way. Instead they are treated in a special way.

Examining the last rule defining norisk_1, for each iteration evaluating tuples satisfying norisk_1, a potentially new X is generated when solving ¬prone(X). Following the negation as failure rule (Clark, 1978), each such negative query is first asked as a positive query. Ideally, in an analogous way to positive queries we prefer that the query is asked with X effectively bound so that an existential query is asked. If this is not the case, then the set of *all* people who are prone must be retrieved. This is avoided in the positive case by using the magic set. For the rules defining prone, a magic set, magic_prone, is in place to capture this bound value and direct the search. However, as indicated above, rule (1) unstratifies the database.

The solution partitions the rules of the program into separate nodes as shown in figure 2. We associate with each node a set of rules, and a set of tuples which are evaluated iteratively using a structured bottom-up computation. Using the rules associated with a node, each successive arc inside the node pictorially delineates the tuples derived at each successive iteration i, $1 \leq i \leq n$.

With reference to our example, Node R contains all the rules except those defining prone, since prone is asked when inferring answers to the negative query ¬prone(X). Node(s) P contain the rules defining prone and magic_prone except for the usual magic_prone fact.

A bottom-up computation proceeds in R until the answers to ¬prone(X), at each iteration, are needed. Con-

trol then passes to the rules in P for a bottom-up computation of prone(X) as required to infer ¬prone(X). In addition to control passing to P, as illustrated by the directed arcs between nodes, a set of tuples which is determined using rule (1) and the tuples generated in R *for that iteration* are sent to P. These constitute the aforementioned missing magic_prone fact for this iteration. The answers to prone(X) are evaluated and control returns to R for the next iteration until saturation.

3 NEW LABELING ALGORITHM

3.1 Motivation

There are a number of drawbacks of the BPR algorithm that we address. Referring back to example 1, two types of generic queries can be asked with respect to the predicate antidote. The first is when the argument is not bound. That is, "list the people for whom an antidote is possible". We can *adorn* (Ullman, 1985) the predicate with an f to indicate that the argument is free as in $\texttt{antidote}^{\texttt{f}}$. The other possibility is for the argument to be bound, as in "will an antidote be effective for so-and-so". This is indicated by a b adornment as in $\texttt{antidote}^{\texttt{b}}$. A compiled approach using magic sets can generically transform a database according to a particular query type (adornment). All the transformed rules of example 2, excepting the fact magic_antidote(petra) can be derived and pre-compiled an the actual query is asked. The only step that must be performed at query time is the initialisation of the magic set by a fact.

Unfortunately, although the BPR algorithm is useful for computing a correct semantics, since it labels *all* the positive body literals, a new query may cause the re-labeling of the database at *run-time*. This does not allow a pre-compiled approach to magic sets, and therefore warrants a solution. We illustrate this by an example, where, for simplicity, we omit the arguments of literals. (The reader should bear in mind that a goal of the form ← ¬p, p doesn't automatically fail, since it may abbreviate a goal ← ¬p(X, Y), p(Y, Z))

Example 4

For the program below, assume the query is ← h and that c is a base predicate. After the magic set transformation, the negative cycle $\texttt{p} \overset{*}{\leftarrow} \texttt{magic\_p} \overset{*}{\leftarrow} \neg\texttt{p}$ is created, rendering the transformed database unstratified and so we apply labeling.

Original	BPR Labeling
h ← ¬p, p	h ← ¬p, p_1
p ← c	p ← c
	p_1 ← c

For databases which do not contain negative literals, a subsequent user query and the addition of the query rule do not imply that the rules for the predicates in the body of the query rule have to be re-compiled. Consider a database which does include negative literals. If the user subsequently asks the query ← ¬p, p, h, this is handled in the usual way by converting the query to ← answer and adding the query rule answer ← ¬p, p, h. However, with the BPR labeling algorithm we *are* forced to re-compile the database even though all the predicates for all combinations of adornments had been compiled beforehand. This is because the database is relabeled as

answer ← ¬p, p_1, h_1
h_1 ← ¬p, p_2
p ← c
p_1 ← c
p_2 ← c

and will now also require new magic rules. The new, efficient labeling algorithm which we present does not suffer from this drawback, as we show.

A desirable property of any labeling algorithm is that it does not label a database that does not contain negative body literals. The BPR labeling algorithm does label positive literals even though they may have nothing to do with a resultant negative cycle. In the worst case, as shown later in example 5, the number of generated labels is exponential in the number of derived predicates.

3.2 The Algorithm

The key concept behind any labeling algorithm is to distinguish the context for constructing magic sets. The BPR algorithm performs this contextual separation by labeling each occurrence of a positive literal p with a unique label. Thus, for magic rules containing negative literals ¬p in their bodies, ¬p is effectively distinguished from positive occurrences of p because it is *not* labeled. Our approach here is to *explicitly* label p when it appears as a negative body literal in a rule r. We do *not* label the positive predicates in r. Instead, we only label the predicates of those positive literals that appear in the defining rules for the new *negatively labeled* predicates. There are typically fewer occurrences of negative body literals than positive literals and so, in general, we expect a dramatic drop in the number of labeled predicates.

The following illustrates the labeling employed by the new algorithm. Assume that the query is ← h and that e is a base predicate. After the magic set transformation, the resultant unlabeled database is unstratified because of the negative cycle $\texttt{a} \overset{*}{\leftarrow} \texttt{b} \overset{*}{\leftarrow} \texttt{magic\_b} \overset{*}{\leftarrow} \neg\texttt{a}$, and so we apply labeling.

function *label*($\mathbf{P^a} \bigcup \mathbf{F}$, $\mathbf{S^a}$)
  construct a maximal stratification $\mathbf{L}_1, \ldots, \mathbf{L}_n$ for $\mathbf{P^a}$
  let $\mathbf{S}_i \in \mathbf{S^a}$ be the sips corresponding to the $\mathbf{L}_i$
  initialise the l_i and s_i, $i = 1, n$ to $\emptyset$
  call *neglabel*
  call *poslabel*
  $\mathbf{P^L} := \bigcup_{i=1}^{i=n} \{\mathbf{L}_i \bigcup l_i\}$; $\mathbf{S^L} := \bigcup_{i=1}^{i=n} \{\mathbf{S}_i \bigcup s_i\}$
return($\mathbf{P^L} \bigcup \mathbf{F}$, $\mathbf{S^L}$)

Figure 3: Labeling algorithm

Original	Efficient Labeling
h ← ¬a, b, c, d	h ← ¬n_a, b, c, d
a ← b	n_a ← b_1
c ← b	c ← b
b ← d	b ← d
d ← e	b_1 ← d_1
	d ← e
	d_1 ← e

There are two stages in the labeling algorithm of figure 3. In the first stage, we *negatively label* those predicates which appear as negative body literals and are part of a negative cycle in the dependency graph of the (unlabeled) magic transformed database $\mathbf{D^M}$. We create new rules defining the newly labeled predicates. In the second stage we *positively label* these new rules and create further new rules for the new positively labeled predicates.

Definition. A predicate n_q in a labeled program $\mathbf{P^L}$ is *negatively labeled* if it was formed by replacing a negative occurrence of q in the unlabeled adorned program $\mathbf{P^a}$.

Definition. A predicate p_k in a labeled program $\mathbf{P^L}$, where k is an integer, is *positively labeled* if it was formed by replacing an occurrence of p in the unlabeled adorned program $\mathbf{P^a}$.

The input to the labeling algorithm consists of the set of the strata $\mathbf{L}_i$, $i = 1, n$, which form a *maximal stratification* of $\mathbf{P^a}$, and their corresponding set of sips $\mathbf{S^a}$.

When we assume a maximal stratification, the predicates defined in each stratum correspond to the predicates comprising a MSCC. The initial adorned program is $\mathbf{P^a} = \bigcup_{i=1}^{i=n} \mathbf{L}_i$ and the associated sips are $\mathbf{S^a} = \bigcup_{i=1}^{i=n} \mathbf{S}_i$. Collectively, the rules and sips for a stratum $\mathbf{L}_i$ are denoted by $\mathbf{L}_i^s$. During execution of the labeling algorithm each stratum $\mathbf{L}_i$ has an associated set of newly constructed rules l_i and their sips s_i. Collectively, the new rules and sips created during the algorithm are denoted by l_i^s, $i = 1, n$. The output of the algorithm is the labeled database $\mathbf{D^L}$ and sips $\mathbf{S^L}$.

The first stage of the algorithm calls *neglabel* in figure 4 which performs the negative labeling. We make use of the following sets and functions.

```
procedure neglabel
  for i := 1 to n do
   for each q ∈ negBodLits(i) do
    if negcycle({q}) then
     replace each q in negative body literals of L_i^s by n_q
   od
   if negcycle(definedIn(i)) then
    add a copy of L_i^s to l_i^s
    replace each q ∈ definedIn(i) in l_i^s with n_q
   fi
  od
end
```

Figure 4: Negative labeling procedure

definedIn(i) is the set of predicates defined in the stratum $\mathbf{L}_i$.

negBodLits(i) is the set of predicates that appear as negative body literals in the stratum $\mathbf{L}_i$.

negcycle ($\mathcal{A}$) returns *true* if a predicate p $\in \mathcal{A}$, where $\mathcal{A}$ is a set of predicates, is part of a negative cycle in the dependency graph corresponding to the (unlabeled) magic transformed program; otherwise it returns *false*.

For the second stage, which uses the *poslabel* procedure of figure 5, we associate a single counter C_i with each stratum, $\mathbf{L}_i$. The following extra functions are used by *poslabel*.

append(p, k) returns the string formed by appending the character string corresponding to the value of the integer expression k to the string "p_" where p is a predicate name.

doLabel(t_j^s) Let p be a predicate defined in $\mathbf{L}_m$. The function *doLabel* replaces each *unlabeled* predicate p that appears as a positive literal in t_j^s by *append*(p, $C_m + 1$). Note that this may include p in the head or body of rules in t_j^s.

depends(i, j) returns true if there exists a path in $\mathcal{G}$ from a predicate defined in $\mathbf{L}_j$ to a predicate defined in $\mathbf{L}_i$; otherwise it returns false.

Proposition 1 *Let* $\mathbf{D^L}$ *be the resultant database after applying the labeling algorithm of figure 3 to* $\mathbf{D}$. *For any query* q, *where* q *is defined in* $\mathbf{D}$, *an instance of* q *is in* $M_{\mathbf{D}}$ *if and only if it is in* $M_{\mathbf{D^L}}$.

The proof is similar to that in (the technical report version of) (Beeri et al., 1987).

```
procedure poslabel
 set each counter C_i to 0
 for i := 1 to n − 1 do
  if negcycle(definedIn(i)) then
     doLabel(l_i^s)
     for j := i − 1 downto 1 such that depends(i, j) do
          make a copy of L_j^s called t_j^s
          doLabel(t_j^s)
          add t_j^s to l_j^s
          C_j := C_j + 1                                  (*)
     od
  fi
 od
end
```

Figure 5: Positive labeling procedure

Proposition 2 *If* $\mathbf{D^L}$ *is the resultant database after applying* label *of figure 3 to a stratified database* $\mathbf{D}$, *and* $\mathbf{D^M}$ *is the resultant database after applying the magic set algorithm (Balbin et al., 1987) (constructing magic rules only for positive literals) to* $\mathbf{D^L}$, *then* $\mathbf{D^M}$ *is stratified.*

For the proof, see (Balbin et al., 1988).

The labeling algorithm creates new rules corresponding to the newly negatively and positively labeled predicates. We now analyse the maximum possible number of these predicates in $\mathbf{D^L}$.

Proposition 3 *Let* $\mathbf{D^L}$ *be the resultant database after applying the labeling algorithm of figure 3 to* $\mathbf{D}$ *and let* $\mathbf{D_1}, \ldots, \mathbf{D_n}$, $n \geq 1$ *be the original maximal stratification used. If* m *is the number of derived predicates in* $\mathbf{D}$ *and* v *is the number of new labeled predicates in* $\mathbf{D^L}$ *then* $v \leq m * n$.

Proof. Let m_i be the number of predicates defined in each $\mathbf{L}_i$, $1 \leq i \leq n$, where the $\mathbf{L}_i$ comprise the maximal stratification before labeling, so that $m = \sum_{i=1}^{i=n} m_i$. Similarly, let v_i be the number of predicates defined in each l_i, $1 \leq i \leq n$, where the l_i are the modified stratum output by the labeling algorithm, so that $v = \sum_{i=1}^{i=n} v_i$. Clearly, the maximum number of negatively labeled predicates is less than or equal to m. The number of positively labeled predicates defined in l_i is equal to the value of m_i at the end of the algorithm. By construction, the maximum value for C_i at the end of the algorithm is $n - i$, $1 \leq i \leq n$. Therefore, the maximum number of positively labeled predicates is

$$
\begin{aligned}
& m_{n-1} * 1 + m_{n-2} * 2 + \ldots + m_1 * (n-1) \\
\leq & m_{n-1} * (n-1) + m_{n-2} * (n-1) + \ldots + m_1 * (n-1) \\
\leq & m * (n-1) \\
\Rightarrow & v \leq m * n
\end{aligned}
$$

Corollary 1 *If the number of rules in* $\mathbf{D}$ *is* s *then the number of new rules in* $\mathbf{D^L}$ *is less than or equal to* $s * n$.

Example 4 (*Revisited*).

Example 4 highlights that no re-labeling is required with the new scheme.

Original	BPR Labeling	Efficient Labeling
h ← ¬p, p	h ← ¬p, p_1	h ← ¬n_p, p
p ← c	p ← c	p ← c.
	p_1 ← c	n_p ← c.

This time if the user subsequently asks the query ← ¬p, p, h, which we handle by converting the query to ← answer and adding the rule answer ← ¬p, p, h, no re-labeling is required with the efficient labeling algorithm. This is because positive body literals in a query rule are not labeled by the algorithm. Therefore, when *all* derived predicates in the database, for all their different adornments, irrespective of whether they appear as positive or negative body literals are pre-compiled before any subsequent user queries, no re-compilation is necessary. This is analogous to the case for databases with only positive body literals and therefore preserves the spirit of the compiled approach. With the BPR labeling algorithm, however, we are forced to re-compile the database, as we described earlier.

The following example highlights the worst case performance of the labeling algorithm.

Example 5.

Original Program	Labeled Program
a ← ¬b, b, ¬c, c, ¬d, d	a ← ¬n_b, b, ¬n_c, c, ¬n_d, d
b ← ¬c, c, ¬d, d	b ← ¬n_c, c, ¬n_d, d
c ← ¬d, d	c ← ¬n_d, d
	c_1 ← ¬n_d, d_1
d ← e.	d ← e
	d_1 ← e
	d_2 ← e
	n_b ← ¬n_c, c_1, ¬n_d, d_1
	n_c ← ¬n_d, d_2
	n_d ← e.

4 CONCLUSION

Magic sets are an efficient query processing strategy for deductive databases containing recursive rules. The main strength of the magic set approach is that it permits a database to be pre-compiled. When magic set algorithms are naively extended to include negation, the resultant database is often unstratified. One step in a solution to this problem involved the BPR labeling algorithm to separate context dependencies. Although the BPR algorithm is adequate for preserving stratification properties, in general, re-labeling is required and therefore, re-compilation for new queries. In addition

the size of the compiled code may be very large due to the exponential number of newly generated predicates.

A new algorithm for labeling a database is presented. If n is the number of levels in the maximal stratification, then, in the worst case, the number of generated labeled predicates is less than or equal to n^2. Typically, we expect this to be linear. This is a significant improvement over the BPR algorithm. Another important advantage is that new queries no longer cause re-labeling since the labeling algorithm is *query independent*. The only step that takes place at query time, analogous to the case for databases not containing negative body literals, is the construction of the query rule. This facilitates the traditional pre-compilation of a database in the spirit of the magic set approach for positive programs. Additionally, because the new algorithm is expressed independently of sips, altering a sip does not require re-labeling (although it will result in different magic rules).

Further optimisations suggested in (Balbin et al., 1987) are equally applicable with the new labeling scheme. For example, if the transformed program contains predicates, p_1, p_2, ..., p_n, the modified bottom-up computation needs only to maintain one internal predicate which represents the tuples satisfying the p_i, $i = 1, n$.

ACKNOWLEDGEMENTS

We thank Graeme Port for typically incisive comments. The second author wishes to thank Jim Baxter and Neil Harrington for their support and encouragement.

REFERENCES

APT, K. R., BLAIR, H. A., & WALKER, A. (1988). *Towards a Theory of Declarative Knowledge*, pages 89–148. In (Minker, 1988).

BALBIN, I., MEENAKSHI, K., & RAMAMOHANARAO, K. (1988). A Query Independent Method for Magic Set Computation on Stratified Databases. Technical report, Department of Computer Science, Royal Melbourne Institute of Technology.

BALBIN, I., PORT, G., & RAMAMOHANARAO, K. (1987). Magic Set Computation for Stratified Databases. Technical Report 3 (Revised), Department of Computer Science, University of Melbourne, Australia.

BALBIN, I. & RAMAMOHANARAO, K. (1987). A Generalisation of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Programming*, 4(3):259–262.

BANCILHON, F. (1985). Naive Evaluation of Recursively Defined Relations. *Proceedings of the Islamadora Conference on Database and AI.*

BANCILHON, F., MAIER, D., SAGIV, Y., & ULLMAN, J. (1986). Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 1–15, Washington, DC.

BANCILHON, F. & RAMAKRISHNAN, R. (1988). *Performance evaluation of data intensive logic programs.* In (Minker, 1988).

BEERI, C., NAQVI, S., RAMAKRISHNAN, R., SHMUELI, O., & TSUR, S. (1987). Sets and Negation in a Logic Database Language (LDL1). In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 21–37, San Diego, California.

BEERI, C. & RAMAKRISHNAN, R. (1987). On the Power of Magic. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 269–283, San Diego, California.

CHANDRA, A. & HAREL, D. (1985). Horn Clause Queries and Generalization. *Journal of Logic Programming*, 2(1):1–15.

CLARK, K. (1978). Negation as Failure. In GALLAIRE, H. & MINKER, J., editors, *Logic and Databases*, pages 293–322. Plenum Press, New York and London.

HAN, J. & LU, H. (1986). Some Performance Results on Recursive Query Processing in Relational Databases. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 533–541, Los Angeles, California.

MINKER, J., editor (1988). *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, Los Altos, California.

ROHMER, J., LESCOEUR, R., & KERISIT, J. (1986). The Alexander Method - A Technique for the Processing of Recursive Axioms in Deductive Databases. *New Generation Computing*, 4(3):273–286.

SACCA, D. & ZANIOLO, C. (1987). Implementation of Recursive Queries for a Data Language based on Pure Horn Logic. In *Proceedings of the 4th International Conference on Logic Programming*, pages 104–135, University of Melbourne.

ULLMAN, J. (1985). Implementation of Logical Query Languages for Databases. *ACM Transactions on Database Systems*, 10(3):289–321.