Magic-sets Transformation in Nonrecursive Systems

Submitted to *Transactions on Database Systems*

Ashish Gupta[*]
Computer Science Department
Stanford University
Stanford, CA 94305, USA.
agupta@cs.stanford.edu

Inderpal Singh Mumick
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA.
mumick@research.att.com

August 1993

1 Introduction

The Problem: Most existing database systems (IBM's DB2, for example) do not support recursive queries. SQL is the universal query language amongst relational databases, and the currently implemented SQL standard (SQL2, [ISO90]) does not provide for recursion. We will call a database system nonrecursive if its query language does not permit recursive queries. Thus, DB2 is a nonrecursive system.

The magic-sets transformation ([BR87, Ull89]) is a general purpose query optimization technique initially proposed for recursive queries in deductive databases. It has been shown that the magic-sets transformation (MST) can be extended to optimize relational SQL queries where SQL has been extended to permit recursion [MFPR90, MPR90, Mum91]. Performance experiments have demonstrated that MST is an invaluable optimization for nonrecursive queries [MFPR90, Mum91]. It is thus imperative that MST be used to optimize queries in a nonrecursive system.

However, there is one problem that must be solved before MST can be used in a nonrecursive system. MST has the undesirable property that it can transform a nonrecursive query into a recursive query, as in Example 1.1 below.

EXAMPLE 1.1 Consider query Q on the nonrecursive program P (We use a letter, such as P, to represent a program comprised of rules $P1, P2, \ldots$):

(Q): ?- $t(0)$.

$(P1)$: $t(X)$:– $s(X, Y)$ & $p(Y)$.

$(P2)$: $s(X, Y)$:– $p(X)$ & $q(X, Y)$.

*Work was supported by an NSF grant IRI-91-16646, an Air Force grant AFOSR-88-0266, and a grant of IBM Corporation.

($P3$): $p(X) :- e_1(X,Y)\ \&\ e_2(Y)$.

($P4$): $q(X,Y) :- e_3(X,Z)\ \&\ e_4(Z,Y)$.

Applying the magic-sets transformation to program P, we derive the program MP below:

($MP1$): $t(X) :- m\_t(X)\ \&\ s(X,Y)\ \&\ p(Y)$.

($MP2$): $s(X,Y) :- m\_s(X)\ \&\ p(X)\ \&\ q(X,Y)$.

($MP3$): $p(X) :- m\_p(X)\ \&\ e_1(X,Y)\ \&\ e_2(Y)$.

($MP4$): $q(X,Y) :- m\_q(X)\ \&\ e_3(X,Z)\ \&\ e_4(Z,Y)$.

($MP5$): $m\_t(0)$.

($MP6$): $m\_s(X) :- m\_t(X)$.

($MP7$): $m\_p(Y) :- m\_t(X)\ \&\ s(X,Y)$.

($MP8$): $m\_p(X) :- m\_s(X)$.

($MP9$): $m\_q(X) :- m\_s(X)\ \&\ p(X)$.

Program MP is recursive since predicate m_p depends upon predicate s (rule $MP7$), s depends upon predicate p (rule $MP2$), and p depends upon predicate m_p (rule $MP3$). The recursion arises because rule $P1$ asks for information to be passed sideways from predicate s into predicate p, while rule $P2$ defines predicate s using predicate p. □

A transformation that produces recursive queries cannot be used in a system that does not support recursive queries. Can we somehow avoid introducing recursion when doing the magic-sets transformation? If recursion could be avoided, the magic-sets transformation would be usable in nonrecursive systems, and thus be a very attractive optimization for all existing relational database systems.

The Solution: We first show that recursion introduced by the magic-sets transformation is bounded [Nau86], and present a simple algorithm for avoiding recursion by replicating all common subexpressions in the original program. The algorithm is called RACS/MST (*Replicate All Common Subexpressions and apply Magic-Sets Transformation*). However, each replication of a common subexpression leads to replicated work, and it would be preferable not to duplicate all common subexpressions.

We then develop a more efficient algorithm for nonrecursive magic-sets transformation: RLCS/MST (*Replicate Limited Common Subexpressions and apply Magic-Sets Transformation*). RLCS/MST does limited replication of common subexpressions before applying the magic-sets transformation. The algorithm can be altered to unfold common subexpressions instead of replicating them.

Relevance: Existing commercial database systems do not support recursive queries. Recursive queries are not needed in many traditional applications, and nonrecursive database systems will continue to be developed. Our work makes it possible to incorporate the magic-sets transformation into nonrecursive systems. In fact, the magic-sets optimization can be built as a preprocessor *on top* of the database where queries are precompiled and then provided to the underlying database. This approach does not impact the database at all.

As proposed in SQL3, [ISO90], SQL is being extended to permit recursion and perhaps most future database systems will support recursion. However, even in recursive systems it is desirable not to transform a nonrecursive query into a recursive one since it is expected that recursive queries will be less efficient to execute than equivalent nonrecursive queries.

We also present an optimization *covered subgoal elimination*, that can be used to eliminate recursive subgoals in programs.

Paper Layout: The paper is organized as follows. We start with some preliminary definitions in Section 2, and consider related work in Section 2.1. A new graphical structure, the *information passing graph* (IPG), is introduced in Section 3. An IPG represents the sideways passing of information (join predicates) between subgoals in a rule together with the dependencies from the subgoals to the head of a rule. An IPG is a very useful tool to reason about recursion introduced by MST. Section 4 presents the RACS/MST algorithm based on replication of *all* common subexpressions, and shows, by deriving a nonrecursive program equivalent to the magic-sets transformed program, that the recursion introduced by MST is bounded. The RLCS/MST algorithm based on *limited* replication of common subexpressions is discussed in Section 5. Section 6 discusses covered subgoal elimination, a technique to eliminate redundant subgoals (joins) from a rule. We end with conclusions in Section 7.

2 Background

In this paper we will write database queries in datalog [Ull89]. Datalog is the language of horn clause logic rules, written as "if-then" rules. A datalog rule is required to be range-restricted: it must define a *finite* relation for the predicate in the head of the rule. In this paper we will concentrate on nonrecursive queries. Mumick [Mum91] shows that nonrecursive datalog programs can be expressed in SQL using views, and that SQL queries can be expressed in datalog extended with duplicates and grouping with aggregation. Thus, the algorithms developed in this paper can be mapped to relational database systems having SQL as their query language.

We will use dependency graphs to represent the dependence between predicates in a datalog program.

Definition 2.1 Dependency Graph (DG): The dependency graph, DG(P), for a datalog program P is a directed graph built as follows: for every ordinary predicate p in program P, there is a node labeled p. There is an edge from node q to node p (q and p may be the same node) if there is a rule of the form:

(r): p :- … & q & ….
□

EXAMPLE 2.1 Figure 1 shows the dependency graph for program P introduced in Example 1.1. □

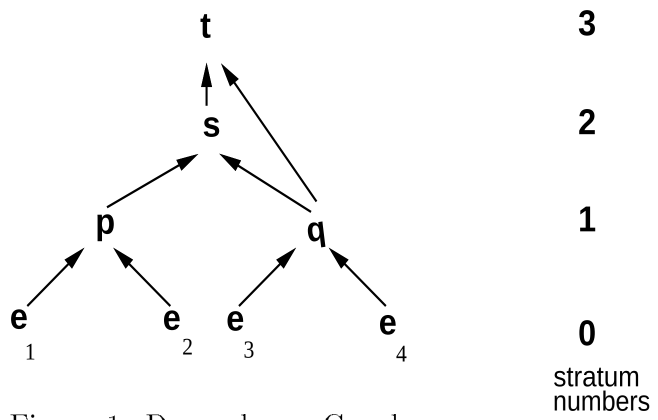

Figure 1: Dependency Graph

A datalog program is recursive if and only if its dependency graph has a cycle, for then the predicates in the cycle are defined in terms of each other. In this paper we concentrate on nonrecursive programs, and assign stratum numbers to predicates as follows:

Definition 2.2 Stratum Numbers: In a nonrecursive program P, the stratum number of a base predicate is 0, and the stratum number of a derived predicate is one more than the largest stratum number of all its children in the dependency graph. $\Box$

Our definition of strata differs from the definition given by Ullman ([Ull89], Chapter 3.6). Given two mutually nonrecursive predicates p and q in a datalog program (without negation), Ullman would place both predicates into the same stratum. However, we place predicates p and q in different strata. If predicate p depends on predicate q, then predicate p is placed in a higher stratum. If predicates p and q do not depend on each other, then it is possible for the predicates to be placed in the same stratum.

We assume the reader is familiar with the concepts of sips, adornments, magic-sets, and conjunctive query containment covered in [Ull89]. A sideways information passing strategy (sips) is a decision on how to pass information sideways in the body of a rule while evaluating the rule. The magic-sets transformation is defined on adorned programs, and is guided by a sips. The adornments are used to identify the bound arguments of each subgoal in the rule. To simplify the treatment in this paper, we will work on unadorned programs, and we will assume we know the bound arguments of each predicate.

The following definition of the magic-sets transformation is as described in [BR87], simplified only to omit the adornments.

Definition 2.3 The Magic-sets Transformation (MST): Given a program P, a query goal q, and a full sips for each rule for program P, the magic-sets transformation derives a magic program MP as follows:

1. Create a new predicate m_p (the magic predicate for p) for each derived predicate p in program P. The arity of m_p is the number of bound arguments of p.

2. For each rule r in program P, add a *modified version* of rule r to program MP. If rule r has head $p(\overline{t}\,)$, where $\overline{t}$ is shorthand for all arguments of p, the modified version is obtained by adding the literal $m\_p(\overline{t^b})$ into the body of rule r as the first subgoal, where $\overline{t^b}$ denotes all bound arguments of $p(\,\overline{t}\,)$.

3. For each rule r in program P with head $p(\overline{t}\,)$, and for each subgoal $q_i(\overline{t_i}\,)$ in its body such that q_i is a derived predicate, add a *magic rule* to program MP. The head is $m\_q_i(\overline{t_i^b})$. The body contains all subgoals that precede q_i in the sips order associated with r, as well as the literal $m\_p(\overline{t^b})$.

4. Create a *seed* fact $m\_q(\overline{c})$, where $\overline{c}$ is the set of constants equated to the bound arguments of the query goal.

□

2.1 Related Work

A problem similar to the one discussed in this paper, is doing magic-sets transformation on datalog programs with stratified negation and ensuring that the negation remains stratified. Similarly, ensuring the same well founded semantics of a program before and after MST, has been treated in [Ros90, KSS91]. In [BPRM91], the authors describe a labeling algorithm and transformation on the programs to ensure that the MST of a stratified program is also stratified. The labeling algorithm identifies all negated predicates that also occur positively and fully replicates the negated instances, guaranteeing that after replication MST does not introduce recursive negation. The intuition for replication is the same as in this paper. The exhaustive replication is similar to the RACS/MST algorithm described in Section 4. However, our method for doing restricted replication could also reduce the number of negated predicates replicated by [BPRM91].

Another alternative for ensuring that the MST of a nonrecursive program is nonrecursive, is to use a SIPS that ensures the condition. The idea of a well-founded SIPS [KSS91] is useful here. Informally, if the SIPS used in the program respects the stratification, then information passing will be in the same direction as the dependencies and therefore MST will not introduce cycles. However, often this SIPS is not the most desirable one, and in general we need a way of producing a nonrecursive MST given an arbitrary SIPS.

The ideas described in [RSS90] can also be used to make the evaluation of the recursive MST more efficient. [RSS90] describes how to determine optimal orders for rule evaluation while computing a recursive program. Their strategy is also used to evaluate programs with stratified negation that can become potentially non-stratified after magic-sets transformation. However, they rely on changing the evaluation strategy of the rules in order to achieve this effect. The rules still remain recursive. By ensuring that the magic-sets transformed program is nonrecursive, we avoid altering the evaluation strategy.

A preliminary report on this work appeared in [GM92]. The RLCS/MST algorithm presented in the paper has been improved (and is different) from the one in [GM92]. We also include proofs, and and describe additional useful properties of the Information Passing Graphs that represent the information passing characteristics of a program.

3 Information Passing Graphs

Dependency graphs capture the "upwards" flow of information from subgoals to the predicates they derive. Information can also be passed sideways from one subgoal into another in the same rule, as specified by a sips. The magic-sets transformation introduces magic predicates to pass information sideways. We define information passing graphs to represent this sideways flow of information for a given sips, in addition to the "upward" flow already represented by a dependency graph.

Rules that have multiple occurrences of the same predicate in the body always lead to recursion after doing magic-sets transformation, since magic-sets transformation tries to pass information from one occurrence of the predicate into another. However, in a non recursive program, we can replicate the definition of a predicate p to get equivalent predicates $p_1, \ldots, p_i$, substitute the multiplle occurrence of p by $p_1, \ldots, p_i$, and thereby derive equivalent rules that do not have repeated subgoals, in the rest of this paper we assume that no two subgoals in any rule in the program have the same predicate. That is, there are no rules of the form:

(a): $q :- S_0 \ \& \ p \ \& \ S_1 \ \& \ p \ \& \ S_2.$

For such programs the Information Passing Graph is defined as follows:

Definition 3.1 Information Passing Graphs (IPG)**:** The information passing graph of a datalog program P, $\mathsf{IPG}(P)$, is a directed labeled graph obtained by extending the dependency graph. The IPG has a *predicate* node for every predicate q in program P, and an additional *information* node iq for every *derived* predicate q in program P. Every predicate node has a stratum number obtained from the dependency graph (Definition 2.2). Information nodes in an information passing graph do not have stratum numbers. The IPG has two types of edges: dependency (**D**) edges denoted by $\rightarrow$ and information (**I**) edges denoted by $\Rightarrow$. Given rules of form r

(r): $q_0 :- q_1 \ \& \ q_2 \ \& \ \ldots \ \& \ q_n.$

the edges of the information passing graph are derived as follows:

1. A dependency edge $q_i \rightarrow q_0$ is derived from every rule of form r.

2. An information edge $iq_0 \Rightarrow iq_j$, is derived from every rule of form r, where q_j is the first derived subgoal in rule r. Such edges are referred to as **II** edges.

3. Given a subgoal q_i in a rule of form r, let q_j be the first derived subgoal (if one exists) after subgoal q_i. The information edge $q_i \Rightarrow iq_j$, is derived from every subgoal q_i in every rule r, provided a derived subgoal follows q_i. Such edges are referred to as **DI** edges.

4. An information edge $iq_0 \Rightarrow q_0$ is derived from every rule of form r. Such edges are referred to as **ID** edges.

A particular **I** or **D** edge can be derived more than once, from different rules. An edge is labeled to indicate all its derivations. An edge label is a set of rule numbers where each rule number uniquely identifies one derivation of that edge. Because every predicate can occur only once in every rule, the rule number along with the head and tail of the edge, is sufficient to identify the predicates that contributed that edge. □

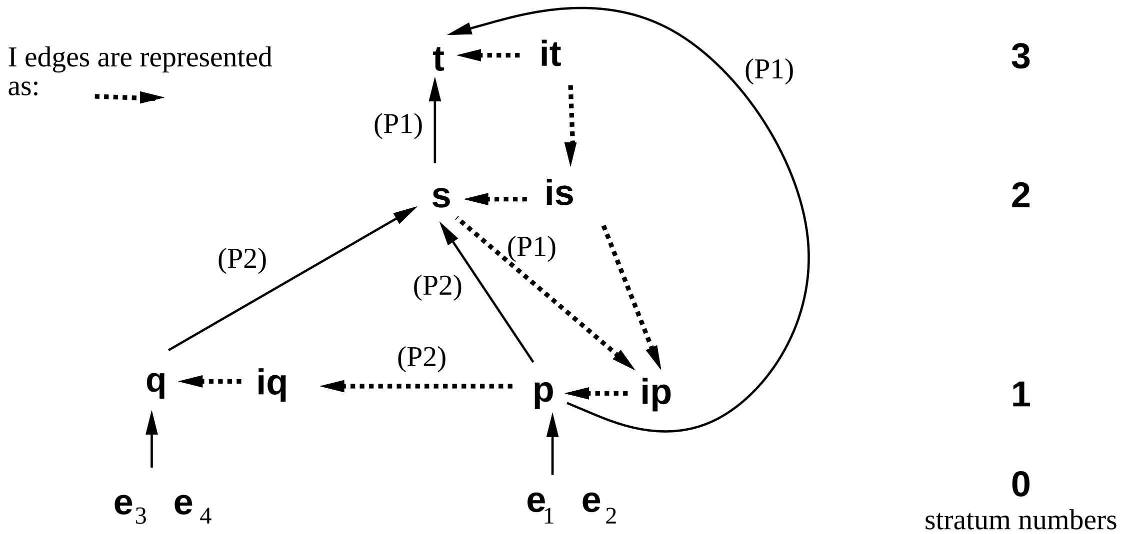

Figure 2: Information Passing Graph for program P of Example 1.1

EXAMPLE 3.1 Figure 2 shows the information passing graph of program P introduced in Example 1.1. Information nodes iq, ip, is and it correspond to the derived predicates q, p, s, and t respectively. In rule $P1$, $p(Y)$ is the first derived predicate after subgoal $s(X,Y)$, deriving the information edge $s \Rightarrow ip$. $s(X,Y)$ and $p(Y)$ are the first and second subgoals respectively in rule $P1$, and contribute the label $(P1)$ to the edge $s \Rightarrow ip$. The edge $is \Rightarrow ip$ is derived from rule $P2$, and has the label $(P2)$. In this paper we will ignore labels on outedges from information nodes, since these labels do not have a bearing on our discussions. □

I edges in an information passing graph represent the sideways flow of bindings between subgoals in a rule, and the downward flow of bindings from the head of a rule into the body of a rule. The magic-sets transformation uses this information passing to derive the magic program MP. We can relate an information node ip in the information passing graph of a program P to the magic predicate m_p in program MP. Paths in $\mathsf{IPG}(P)$ then represent paths in the dependency graph of program MP.

The following theorems state some properties of the IPG of a program.

Theorem 3.1 *Let MP be the program obtained by magic-sets transformation of a given nonrecursive program P. Let $\mathsf{DG}(MP)$ be the dependency graph of program MP, and let $\mathsf{IPG}(P)$ be the information passing graph of program P. Relate the node for a magic predicate m_p in $\mathsf{DG}(MP)$ to the information node ip in $\mathsf{IPG}(P)$. Then, there is a path between two nodes in $\mathsf{IPG}(P)$ if and only if there is a path between the corresponding nodes in $\mathsf{DG}(MP)$.* □

Proof: It is easily seen that each edge in $\mathsf{IPG}(P)$ occurs in $\mathsf{DG}(MP)$. Therefore for every path in the information passing graph, there is a path in the dependency graph.

We now prove that for every edge in $\mathsf{DG}(MP)$, there is a path in $\mathsf{IPG}(P)$. Note that there is a 1-1 correspondence between the nodes of the two graphs. The edges in the dependency graph of the original program are preserved in both $\mathsf{IPG}(P)$ and $\mathsf{DG}(MP)$, i.e. the **D** edges. The additional edges in $\mathsf{DG}(MP)$ as per Definition 2.3, are as follows:

$m\_p \rightarrow p$**:** A corresponding **ID** edge, $ip \Rightarrow p$, exists in $\mathsf{IPG}(P)$.

$q \rightarrow m\_p$**:** One such edge is in $\mathsf{DG}(MP)$ for each subgoal q that precedes subgoal p in a rule r in program P and passes information to p as directed by the SIPS. Consider one such edge $s \rightarrow m\_p$. Let t be the first subgoal that s passes information to in rule r. $\mathsf{IPG}(P)$ will have a **DI** edge $s \Rightarrow it$. Using the **ID** edge $it \Rightarrow t$, we get a path from s to t. This reasoning can be used to make a path to all subgoals that occur between t and p and receive information from s. Path $s \rightarrow\!\!\!\rightarrow p$ therefore exists and is composed of a combination of **DI** and **ID** edges.

$m\_p \rightarrow m\_q$**:** One such edge is in $\mathsf{DG}(MP)$ for each subgoal q that occurs in a rule r that defines predicate p. The IPG has an **II** edge $ip \Rightarrow is$, where s is the first subgoal in rule r. An **ID** edge connects node is to s in the IPG. As argued in the previous case there is a path from s to all subsequent subgoals in rule r that receive information from s. Using these three facts, we can build a path from ip to the magic predicate of any subgoal that receives sideways information in rule r.

□

Corollary 3.1 *The magic-sets transformation of a nonrecursive program P is recursive if and only if the information passing graph of program P has a cycle.* □

Proof: Follows from Theorem 3.1. □

Lemma 3.1 *If the dependency graph $\mathsf{DG}(P)$ contains the path $u \rightarrow\!\!\!\rightarrow p$, then $\mathsf{IPG}(P)$ contains the path $ip \rightarrow\!\!\!\rightarrow iu$.* □

Proof: By induction over the levels below p. A full sips is assumed. □

We now define some concepts and then state some properties of the cycles in an IPG of a program.

Definition 3.2 Simple Cycle: A simple cycle is a cycle in which no node is repeated. □

A node in an information passing graph that is not an information node will be referred to as a predicate node.

Lemma 3.2 *Every cycle in the information passing graph of a nonrecursive program must have at least one predicate node.* □

Proof: A cycle that consists of only information nodes has only **II** edges. A **II** edge $ip \Rightarrow iq$ implies that predicate q occurs in a rule defining predicate p i.e. there is an edge $p \rightarrow q$ in the dependency graph of the original program. Thus, a cycle in $\mathsf{IPG}(P)$ that consists of only **II** edges implies a cycle in $\mathsf{DG}(P)$. Because we only consider nonrecursive programs, such a condition can never arise. □

Lemma 3.3 *Every cycle in the information passing graph of a nonrecursive program P must have at least one information node.* □

Proof: Consider a cycle C that consists of just predicate nodes and therefore only of **D** edges. Such a cycle would imply that there is a cycle in the dependency graph of program P, which is not possible because P is nonrecursive. □

Lemma 3.4 *Every cycle in the information passing graph of a nonrecursive program P must have at least one predicate node x such that the information node ix is also in the same cycle.* □

Proof: Consider a cycle C in $\mathsf{IPG}(P)$. We consider two possible cases of the length of cycle C.

Length of $C = 2$: By Lemmas 3.3 and 3.2, C has one information node and one predicate node. The outedge from the information node ix therefore has to be to the predicate node. By Definition 3.1, such an outedge can go only to the predicate node x.

Length of $C = n$, $n > 2$: Let ia be an information node in cycle C. If the outedge from ia is to a predicate node, we are done. If not, then the outedge has to be to another information node ib. Let us then consider the information node ib. The same argument as in the case of ia, applies here. Because cycle C has length n, in at most $n-1$ steps we either encounter node ia again or a predicate node. By Lemma 3.2 there has to be predicate node in C. Therefore, from some information node ix we will eventually reach a predicate node via an outedge. By Definition 3.1, this predicate node has to be x. □

Definition 3.3 Small predicate and Small Node: Given a program P and a set of predicates $s = \{p_1, \ldots, p_n\}$, that occur in the program, the predicates in set s with the least stratum number are called the *small predicates* in set s. Note that not all the predicates in program P need to be in the set s.

Let t be the set of predicates appearing in a cycle C in $\mathsf{IPG}(P)$. A predicate node of C corresponding to a small predicate in set t is called a *small node*. □

Definition 3.4 Flex Node: Consider a cycle C. A predicate node p is said to be a flex node if the following two conditions are satisfied:

1. Cycle C contains the information node ip, and
2. Let e_o and e_i be two edges in cycle C such that e_o is the outedge from node p, and e_i is the inedge into node ip, Then, either the label on the outedge e_o has at least one member that does not occur in the label on the inedge e_i, or the label on the inedge e_i has at least one member that does not occur in the label on the outedge e_o.

□

The flex node in a cycle need not correspond to one of the small nodes involved in the cycle.

Definition 3.5 Cycle Stratum Number (CSN): The cycle stratum number of a cycle C, $\mathsf{CSN}(C)$, is the least stratum number of all the flex nodes in cycle C. □

Cycle Stratum numbers for cycles in information passing graphs are well defined since, as Theorem 3.2 will soon show, every cycle has a flex node.

Cycle Identifier	Elements in Order of Traversal	Flex node(s)	CSN
$C1$	p, iq, q, s, ip, p	p	1
$C2$	p, s, ip, p	p	1

Table 1: Simple cycles.

EXAMPLE 3.2 Table 1 lists the simple cycles in the information passing graph shown in Figure 2. Cycle $C1$ has three predicate nodes: p, q, and s. Nodes p and q are in stratum 1, and node s is in stratum 2. The cycle stratum number for both cycles is 1 because node p is flex in both cycles and has stratum number 1. □

The following properties of cycles in information passing graphs are useful.

Lemma 3.5 *Consider the small nodes in a given cycle C. For every small node x, cycle C has the edge $ix \Rightarrow x$.* □

Proof: Similar to the proof of Lemma 3.4. Consider a predicate node a in cycle C such that a is a small node in C. By Definition 3.1, the inedge to a can either be (1) from the information node ia or (2) from a predicate node b. In case (1) we are done. In case (2) the inedge from b to a would have to be a **D** edge. However, then node b would have lower stratum number than a leading to a contradiction (because then a would not be one of the small nodes in C). □

Theorem 3.2 *Every cycle in an information passing graph has at least one flex node.* □

Proof: We discuss a proof based on an induction on the number of information nodes (inodes) in the cycle. The proof can also be done by induction on the length of the cycle.

Base Case: one inode in the cycle C: Consider the case that there is only one inode ip in a cycle. For a node to be flex in a cycle, its inode should also be present in the cycle. Therefore p is the only possible flex node in C. Suppose that node p is not flex. The only node to which node ip can pass information is the node p (because outedges from ip can go to either p or to another inode). In addition the only node that passes information to node ip is a predicate node v. In addition, the only outedge from node p can be a **D** edge to the head s of the rule r containing p as a subgoal because p cannot have an outgoing **I** edge (an **I** outedge from p to the solitary inode ip would require multiple occurrences of p in the same rule which is not possible). The cycle therefore has a path of the form:

$v \Rightarrow ip \Rightarrow p \rightarrow s.$

For p not to be flex, the label on edge $p \rightarrow s$ is the same as the label on edge $v \Rightarrow ip$, say $\{r\}$. Therefore both v and p occur as subgoals in rule r and consequently $\mathsf{SN}(s) > \mathsf{SN}(v)$. For the cycle C to be complete there is also a path from s to v. This path consists only of **D** edges because an **I** edge requires an inode as either the source or destination and the only inode in C, ip, does not occur on the path from s to v (but on the path from v to s). A path consisting on only **D** edges from from s to v, given that $\mathsf{SN}(s) > \mathsf{SN}(v)$, would imply that P is recursive.

We therefore get a contradiction if we assume that p is not flex with respect to the cycle.

Inductive Case: $n+1$ inodes in the cycle C: We show that cycle C has a flex node if a ceratin cycle C' that has $\leq n$ inodes has a flex node. By the inductive hypothesis we know that C' has a flex node and therefore, we conclude that C has a flex node. The construction of C' given C is as follows:

Consider a node x such that x is a small node and node ix also occurs in cycle C. By Lemma 3.5 there is such an x. If node x is flex, then cycle C has a flex node. Let's consider the case where node x is not flex. We will now construct a cycle C' that is exactly like cycle C but does not have nodes ix and x and therefore has fewer inodes than C. We are guaranteed that C' has a flex node. The same node will therefore be flex in cycle C also.

One of the following four (exhaustive) cases for the edges into ix and out of x can arise if x is not flex in C. In each case, since x is not flex, the labels on edges into ix and out of x are identical.

$y \Rightarrow ix \Rightarrow x \Rightarrow iz$: Because x is not flex, the inedge to node ix is because of the same rule as the outedge from x. Therefore this sequence of edges can occur in cycle C only if there is a rule r in the original program of the form:

$$(r): \quad p :- \ldots y \;\&\; \ldots \;\&\; x_{-1} \;\&\; x \;\&\; x_{+1} \;\&\; \ldots \;\&\; z \ldots .$$

Where x is the first derived predicate to which y passes information. And similarly z is the first derived predicate to which x passes information. In addition C contains a path $iz \twoheadrightarrow y$. However, consider a new program that is exactly the same as the original program except that rule r is replaced by rule r'.

$$(r'): \quad p :- \ldots y \;\&\; \ldots \;\&\; x_{-1} \;\&\; x_{+1} \;\&\; \ldots \;\&\; z \ldots .$$

Rule r' does not have subgoal x any longer. Predicate y therefore passes information directly to z by the same SIPS as before, i.e. the sequence of edges $y \Rightarrow ix \Rightarrow x \Rightarrow iz$ gets replaced by the edge $y \Rightarrow iz$. The path $iz \twoheadrightarrow y$ does not contain x and hence stays the same (cycle C is a simple cycle and therefore no node occurs twice in C). Therefore we get a new cycle C' that has one inode less than C. The cycle C' is guaranteed to have a flex node by the induction hypothesis. Because the mapping from the nodes, and edges of cycle C' to C is uniquely defined (the identity mapping), the flex node in cycle C' is the flex node for the original cycle C.

$iy \Rightarrow ix \Rightarrow x \Rightarrow iz$: Because x is not flex, the inedge to node ix is because of the same rule as the outedge from x. Therefore this sequence of edges can occur in cycle C only if rule r is of the form:

$$(r): \quad y :- \ldots \;\&\; x_{-1} \;\&\; x \;\&\; x_{+1} \;\&\; \ldots \;\&\; z \ldots .$$

Where x is the first derived predicate to which y passes information. Consider a new program that is exactly the same as the original program except that rule r is replaced by rule r'.

$$(r'): \quad y :- \ldots \;\&\; x_{-1} \;\&\; x_{+1} \;\&\; \ldots \;\&\; z \ldots .$$

Rule r' does not have subgoal x any longer. Predicate y therefore passes information directly to z by the same SIPS as before. The sequence of edges $iy \Rightarrow ix \Rightarrow x \Rightarrow iz$ gets replaced by the edge $iy \Rightarrow iz$. The path $iz \twoheadrightarrow iy$ stays the same because that does not contain x. By a similar argument as before, cycle C has a flex node.

$y \Rightarrow ix \Rightarrow x \rightarrow z$: This sequence of edges can occur in cycle C only if rule r is of the form:

(r): $\quad z$:- $\ldots y$ & $\ldots$ & x_{-1} & x & x_{+1} & $\ldots$.

Consider a new program that is exactly the same as the original program except that rule r is replaced by rule r'.

(r'): $\quad z$:- $\ldots y$ & $\ldots$ & x_{-1} & x_{+1} & $\ldots$.

Predicate y does not pass information to x any longer and the sequence of edges $y \Rightarrow ix \Rightarrow x \rightarrow z$ gets replaced by the edge $y \rightarrow z$. As before, the path $z \rightarrow\!\!\!\rightarrow y$ stays the same, and we can argue that cycle C has a flex node.

$iz \Rightarrow ix \Rightarrow x \rightarrow z$: This sequence of edges can occur in cycle C only if rule r is of the form:

(r): $\quad z$:- $\ldots$ & x_{-1} & x & x_{+1} & $\ldots$.

Where x is the first derived node that receives information sideways from z. Consider a new program that is exactly the same as the original program except that rule r is replaced by rule r'.

(r'): $\quad z$:- $\ldots$ & x_{-1} & x_{+1} & $\ldots$.

This new program contains a cycle where the sequence $iz \Rightarrow ix \Rightarrow x \rightarrow z$ is replaced by the edge $iz \Rightarrow z$. As before, this new cycle has a flex node that can be mapped, by the identity mapping, to a node in cycle C.

$\square$

Lemma 3.6 *Consider a program P and its information passing graph* IPG*(P). Let p be one of the smallest flex nodes in a cycle C, and let the information node ix appears on the path $ip \rightarrow\!\!\!\rightarrow p$ in cycle C. Then, the predicate node x occurs on the path $ix \rightarrow\!\!\!\rightarrow p$.* $\square$

Proof: Given a cycle C, let q either be one of the smallest flex nodes, or a descendant of one of the smallest flex nodes. Let the path $iq \rightarrow\!\!\!\rightarrow q$ be a part of cycle C. We prove, by induction on the stratum number of node x, that if the information node ix appears on the path $iq \rightarrow\!\!\!\rightarrow q$, then the predicate node x must appear on the path $ix \rightarrow\!\!\!\rightarrow q$.

base case, Stratum of node $x = 1$: The only possible outedge from ix is $ix \Rightarrow x$; hence x appears on the path $ix \rightarrow\!\!\!\rightarrow q$.

induction case, Stratum of node $x = n + 1$: The outedge from ix could be $ix \Rightarrow x$, or $ix \Rightarrow iy$. In the first case, the statement stands proven. In the second case, the edge $ix \Rightarrow iy$ must be derived from a rule of the form:

(u): $\quad x$:- $\ldots$ & y & $\ldots z \ldots$.

Rule u ensures that y is in a lower stratum than x. Hence, by the inductive hypothesis, node y appears on the path $iy \rightarrow\!\!\!\rightarrow q$. Further, since y is not flex, the outedge from y must be labeled with rule u. Consequently, there are two possible outedges from node y: (1) $y \rightarrow x$, in which case x appears on the path $ix \rightarrow\!\!\!\rightarrow q$, or (2) $y \Rightarrow iz$, in which case z must occur in the path $iz \rightarrow\!\!\!\rightarrow q$ by induction. Also, z is not flex, hence the outedge from z can either go into x, or into the next subgoal of rule u. Eventually, the outedge from the last subgoal of rule u must go into node x. Thus, x must be on the path $ix \rightarrow\!\!\!\rightarrow q$. $\Box$

Theorem 3.3 *Consider a program P and its information passing graph $\mathsf{IPG}(P)$. Let p be one of the smallest flex nodes in a cycle C, and let x be any predicate node on the path $ip \rightarrow\!\!\!\rightarrow p$ in cycle C. Then, (1) node x is a descendant of node p in $\mathsf{DG}(P)$, and (2) the information node ix occurs on the path $ip \rightarrow\!\!\!\rightarrow x$.* $\Box$

Proof: We prove, by induction on the length of paths, that the theorem statement is true for all paths of the form $iq \rightarrow\!\!\!\rightarrow q$, where q is either one of the smallest flex nodes, or a descendant of one of the smallest flex nodes.

base case, length $= 2$: The complete path is $iq \rightarrow\!\!\!\rightarrow q$, and the theorem statement is trivially true.

induction case, length $= n + 1$: The first edge in the path must be an **II** edge of the form $iq \Rightarrow ix$, for some information node ix, derived from a rule of the form:

(t): q :- $\ldots$ & x & $\ldots y \ldots$.

Since x is a descendant of q, x must be a descendant of one of the smallest flex nodes. Now, by Lemma 3.6, x appears on the path $ix \rightarrow\!\!\!\rightarrow q$. By induction, the theorem statement is true for all nodes on the subpath $ix \rightarrow\!\!\!\rightarrow x$. Further, since x is a descendant of q, and x is not flex, the outedge from x must be labeled with rule t. There are two possibilities for the outedge:

$x \rightarrow q$: The entire path $iq \rightarrow\!\!\!\rightarrow q$ thus satisfies the theorem statement.

$x \Rightarrow iy$: The node y is the next derived node in rule t, and is thus a descendant of q. The path $iq \rightarrow\!\!\!\rightarrow q$ can be short circuited be considering a program where rule t is modified by eliminating the subgoal x, creating a shorter path $iq \rightarrow\!\!\!\rightarrow q$. The shorter path satisfies the theorem statement due to the inductive hypothesis.

$\Box$

Using Theorem 3.3 and Lemma 3.6, we infer that the path $ip \rightarrow\!\!\!\rightarrow p$ would be of even length.

Theorem 3.4 *Let C be a cycle with p as one of the smallest flex node. For every derived predicate u that is a descendant of p, there is a cycle C' (C' could be the same as C) such that p is one of the smallest flex nodes in C' and nodes u and iu are in C' and lie on the path $ip \rightarrow\!\!\!\rightarrow p$.* $\Box$

Proof: Consider Cycle C in which p occurs as small flex node. Given u, a derived descendant of p, we construct a cycle C' such that u occurs in C' and p is a small flex node in C'.

If p is a small flex node in C, then by Definition 3.4 node ip is in C and there is a path $ip \rightarrow\!\!\!\rightarrow p$ in C. The incoming edge into ip and the outgoing edge from p have at least one different element in their labels. We use the old path $p \rightarrow\!\!\!\rightarrow ip$ and construct a new path $ip \rightarrow\!\!\!\rightarrow p$ that involves u such that no node on this new path is flex. We ensure that p stays flex by keeping the incoming edge to ip and the outgoing edge from p in C' the same as in C.

We prove that there is a path $ip \rightarrow\!\!\!\rightarrow p$ involving u such that no node on the path is flex, by induction on the level at which u is a descendant of p *i.e.* the difference in $\mathsf{SN}(u)$ and $\mathsf{SN}(p)$.

u is an immediate descendant of p *i.e.* there is a rule r that defines p and uses u in its body and $\mathsf{SN}(p) = \mathsf{SN}(u) + 1$. There is a path from ip to iu. This can be proven by induction on the subgoal position of u in r.

- If u is the first subgoal then by definition of the IPG there is an arc $ip \Rightarrow iu$ and $u \rightarrow p$ such that both edges have the label $\{r\}$.
- If u occurs in subgoal position i then consider the first derived subgoal v before u in rule r in position $i-k$. By the inductive hypothesis, there is a cycle C'' that involves v in which p is a small flex node. By the given SIPS, subgoal v passes information to iu contributing the arc $v \Rightarrow iu$ and therefore completing the path from ip to iu. The edge $u \rightarrow p$ completes the path from ip to p and resulting in cycle C'. Both v and u are non-flex in this cycle (all other nodes/edges in C' are the same as that in C'').

u is a descendant of p at level i *i.e.* $\mathsf{SN}(p) = \mathsf{SN}(u) + i$. Say the immediate ancestor of u is v. Therefore, $\mathsf{SN}(v)$ is greater than $\mathsf{SN}(u)$. By the inductive hypothesis, there exists a cycle C_o such that p is the least flex node in C_o and v is part of C_o. As in the base case, we can show that we can extend the path $iv \Rightarrow v$ to include u and iu such that u is not flex (by induction over the subgoal position of u in the rule that defines v). □

Theorem 3.5 *Let p be a predicate that occurs in program P as a subgoal exactly once. Then, the predicate node p cannot be a flex node in any cycle in* $\mathsf{IPG}(P)$. □

Proof: Say p occurs in rule $r1$. All inedges to the information node ip will be labeled $r1$. Similarly all outedges from predicate node p will be labeled $r1$. Therefore, node q cannot be flex (Definition 3.4). □

Due to the above theorem, note that if p is not flex with respect to a cycle C, and ip also occurs in that cycle, and there are no flex nodes on the path $ip \rightarrow\!\!\rightarrow p$, then replacing the path $ip \rightarrow\!\!\rightarrow p$ by a single edge $\Rightarrow$ with the same label as the inedge to ip (or the outedge from p), does not affect the flex nodes in cycle C. This is because the inedges and outedges from all the (possibly) flex nodes in C stay the same.

4 RACS/MST

This section discusses a simple method to modify a program so that its magic-sets transformation is guaranteed to be nonrecursive. The method relies on common subexpression elimination. If a derived predicate is used in two subgoals, the derived predicate is said to form a common subexpression. Nonrecursive programs can be classified into two types depending on the presence or absence of common subexpressions.

Definition 4.1 Hierarchical Program: A nonrecursive program with no common subexpressions is said to be hierarchical. □

Definition 4.2 Directed Acyclic Graph Program: A nonrecursive program with common subexpressions is said to be a directed acyclic graph (dag) program. □

Lemma 4.1 *Consider a rule r in the hierarchical program P:*
(r): q :- q_1 & ... & q_n.
Let P^q represent the minimal subset of program P that is sufficient to define predicate q. Given rule r, no information node or predicate node for a derived predicate occurs in the information passing graphs of programs P^{q_i} and P^{q_j}, where $1 \leq i, j \leq n$, and $i \neq j$.
□

Proof: Say that node s occurs in the information passing graphs of both programs P^{q_i} and P^{q_j}. If s is an information node it, then node t must also occur in both IPGs. This is because if node it occurs in $\mathsf{IPG}(P^{q_i})$, then there is a path from it to q_i and that path necessarily includes predicate node t. Therefore we assume that s is a predicate node. Some predicate node s can occur in both information passing graphs if one of the following three cases is true:

1. Node q_i occurs in $\mathsf{IPG}(P^{q_j})$: q_i then occurs as a subgoal in at least two rules in program P: rule r and a rule involved in the derivation of q_j.

2. A different Node q_j occurs in $\mathsf{IPG}(P^{q_i})$: Similar to Item 1.

3. Node $s \notin \{q_i, q_j\}$ occurs in both IPGs: In this case either s, or one of the nodes on the path from s to q_i, occurs as a subgoal in at least two rules.

In each of the three cases there is a contradiction because P was a hierarchical program and had no common subexpressions. □

Theorem 4.1 *Let P be a hierarchical program, and let MP be the program derived by a magic-sets transformation of program P. Then program MP is nonrecursive.* □

Proof: We prove that a hierarchical program has an acyclic information passing graph by induction over the number of strata, k, in the program.
Outer Induction: Base Case, $k = 1$. If all rules are defined purely in terms of base predicates, then the information passing graph of the program is the same as the dependency graph and it is true that there are no cycles.
Outer Induction: Inductive Step. Consider a program P' that has $k + 1$ strata. We do an induction over the number of rules in the $k + 1^{st}$ stratum to prove that the IPG has no cycles.
Inner Induction: $n = 1$. Let P be a hierarchical program that has k strata. P therefore has an acyclic IPG. Let rule r be the first rule of stratum $k+1$ added to program P to result in a program P'.

(r): q :- q_1 & ... & q_n.
Each of $q_1 \ldots q_n$ are distinct because r is part of a hierarchical program P. The information passing graphs of programs $P^{q_i} \ldots P^{q_n}$ are mutually disjoint by Lemma 4.1. Let us derive the information passing graph of program P' given rule r and $\mathsf{IPG}(P^{q_1}) \ldots \mathsf{IPG}(P^{q_n})$. Nodes q and iq are introduced corresponding to predicate q and its magic predicate respectively, and the following edges are added:

1. $q_i \to q$, for every subgoal q_i.

2. $iq \Rightarrow q$, for predicate q in the head or rule r.

3. $q_i \Rightarrow iq_j$, if subgoal q_j is the first subgoal that receives information from q_i in r.

4. $iq \Rightarrow iq_1$, assuming that q_1 receives sideways information from q in rule r.

We argue that these edges do not introduce any cycles in $\mathsf{IPG}(P)$. Each of these edges involves one of the nodes q, iq, q_i, or iq_i. We now argue that none of these nodes can become part of a cycle, hence proving that the new edges will not introduce any cycles and that $\mathsf{IPG}(P')$ is acyclic.

- Node q: Node q has only inedges because $\mathsf{SN}(q) = k + 1$, and therefore q cannot be a subgoal in a rule in a stratum $\leq k$.

- Node iq: There are no inedges to node iq from any of the predicates in $\mathsf{IPG}(P')$ and therefore, cannot be part of a cycle.

- Node iq_i: Node iq_i has no inedges in $\mathsf{IPG}(P^{q_i})$. The only inedge into iq_i in $\mathsf{IPG}(P')$ is because of a subgoal $q_j, j < i$ in rule r. However for there to be a cycle involving edge $q_j \Rightarrow iq_i$, there has to be a path from node iq_i to node q_j. However every node q_j is added to the IPG when rule r is added (every subgoal in program P' occurs just once). Given that nodes q and iq cannot be part of any cycle, the only incoming edges to q_j can be from subgoals that occur in positions $< j$, none of which is the same as predicate q_i. Hence, a path from iq_i to q_j cannot exist. Hence no inedge, $q_j \Rightarrow iq_i$, can be a part of a cycle. Therefore, node iq_i cannot be in any cycle.

- Node q_i: The outedge $q_i \rightarrow q$ cannot be part of a cycle because node q is not involved in a cycle . The only other outedge from q_i is to $iq_j, j \geq i$. We have already argued that no iq_i can be a part of a cycle. Therefore q_i cannot be a part of any cycle.

Inner Induction: Inductive Step. Say program P' already has n rules and the $n+1^{st}$ rule is added. The arguments made to prove the the IPG remains acyclic on adding the first rule in stratum $k+1$ apply in this case too.

Hence we prove that adding a rule to program P does not make its IPG recursive. $\Box$
Theorem 4.1 suggests the following algorithm for applying the magic-sets transformation to a dag program so as to derive a nonrecursive magic program.

Algorithm 4.1 RACS/MST: *Given a dag program P, transform program P as follows:*

1. *Remove all common subexpressions in program P by replicating any predicate used in two or more subgoals. Let P_h be the resulting hierarchical program.*

2. *Apply magic-sets transformation to program P_h. Let MP be the resulting magic program.*

$\Diamond$

Algorithm 4.1 is named RACS/MST, short for *Replicate All Common Subexpressions and apply Magic-Sets Transformation*. Theorem 4.1 guarantees that the output program *MP* of Algorithm 4.1 is nonrecursive.

Corollary 4.1 *Algorithm 4.1 derives a nonrecursive program.* $\Box$

EXAMPLE 4.1 We will apply Algorithm 4.1 to the program P introduced in Example 1.1. Predicate p is the only common subexpression, and it is replicated into two predicates: p and p_1. The resulting program P_h is:

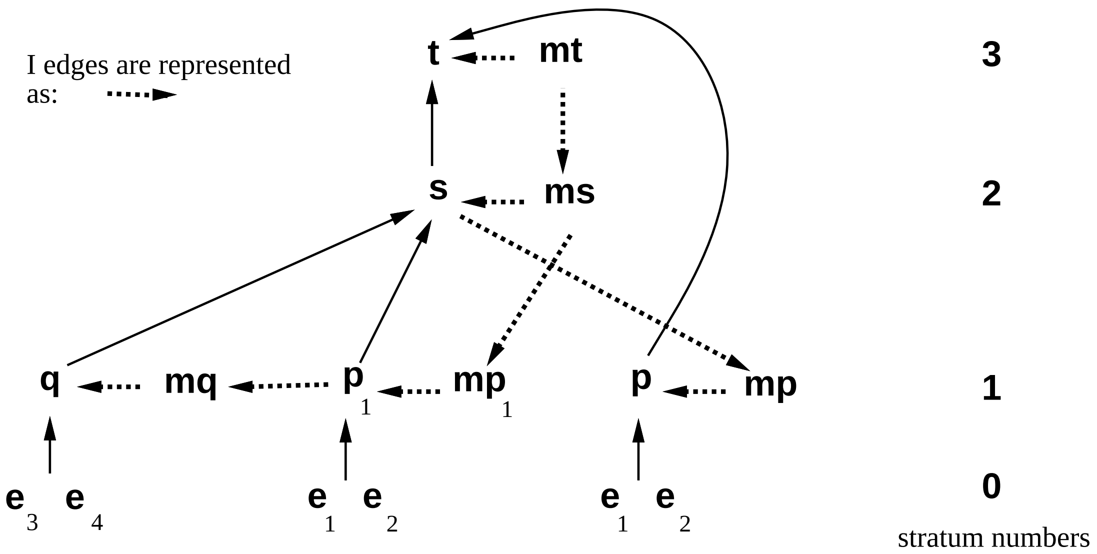

Figure 3: Information Passing Graph of program P_h

(Q_h): ?- :– t(0).

(P_h1): $t(X)$:– $s(X,Y)$ & $p(X)$.

(P_h2): $s(X,Y)$:– $p1(X)$ & $q(X,Y)$.

(P_h3): $p(X)$:– $e_1(X,Z)$ & $e_2(Z)$.

(P_h4): $p1(X)$:– $e_1(X,Z)$ & $e_2(Z)$.

(P_h5): $q(X,Y)$:– $e_3(X,Z)$ & $e_4(Z,Y)$.

Figure 3 shows the information passing graph of program P_h. Since $\mathsf{IPG}(P_h)$ is acyclic, the magic-sets transformation of program P_h is nonrecursive (Theorem 3.1).
□

Algorithm 4.1 implies the following theorem.

Theorem 4.2 *Let P be a dag program, and let MP be the program derived by a magic-sets transformation of program P. Then if program MP is recursive, the recursion is bounded [Nau86].* □

Since every nonrecursive program is either hierarchical or dag, RACS/MST can be used to apply the magic-sets transformation in a nonrecursive system. However, it is not desirable to replicate all common subexpressions in a program, since each replication causes computation to be repeated. Can we do better than RACS/MST? Can we avoid replicating all common subexpressions?

5 RLCS/MST

We now describe an algorithm that does limited common subexpression replication on a nonrecursive program P in order to yield another nonrecursive program whose magic-sets transformation is nonrecursive. We use the notion of flex nodes to restrict the predicates that are replicated in order to break cycles.

5.1 Intuition

The following example illustrates the intuition behind replicating flex nodes:

EXAMPLE 5.1 Consider rules $P1$ and $P2$ from program P in Example 1.1:

$(P1)$: $t(X)$:- $s(X,Y)$ & $p(Y)$.

$(P2)$: $s(X,Y)$:- $p(X)$ & $q(X,Y)$.

p is flex in the cycle $s \Rightarrow ip \Rightarrow p \rightarrow s$ as described in Example 3.2. Intuitively, this is because information comes into one instance of p ($s \Rightarrow ip$ in rule $P1$) but goes out of a different instance ($p \rightarrow s$ in rule $P2$). Replacing p in one of the rules by a different predicate p' would result in the information coming into p but going out of p' (or vice-versa), thereby breaking the cycles that depend on p. The definition of a flex node captures the fact that two instances of p were involved in creating the cycle, by requiring the inedge and outedges to be due to different rules.

If a node is not flex with respect to a cycle, then we know that the same instance of the predicate both receives the incoming information and sends the outgoing information. Replicating such an instance then does not affect the information passing and is therefore useless from the perspective of breaking cycles. □

5.2 The Algorithm

The RLCS/MST Algorithm 5.1 selectively replicates flex nodes in order to break cycles. The cycles are considered in increasing order of their CSN (Definition 3.5). This order ensures that once a predicate is replicated, none of its subgoals introduce any cycles that cannot be broken by considering the old IPG. We thus avoid rebuilding the IPG after replication of a flex node, and need to derive the simple cycles only once. However, the set of simple cycles in the IPG do need to be modified after replication of a flex node. The procedure modify (Algorithm 5.3) does the required manipulation of the original set of cycles. The procedure define_pred (Algorithm 5.2) is used to replicate nodes.

The procedure define_pred replicates the node for predicate p by replacing the predicate p by a new predicate p', that is then defined by the same rules as those defining p. Further, every IDB predicate used in the rules defining p', is itself replicated (using define_pred recursively). This means that all the IDB predicates used to define p' are unique to the rules for p'. Replicating all the IDB predicates involved in the rules for p' ensures that the new rules do not introduce any cycles that are not already represented in the set of cycles $\mathcal{C}$. By "represented" in set $\mathcal{C}$, we mean that every cycle introduced by replication is such that it is guaranteed to be broken when the cycles in $\mathcal{C}$ are broken. Note that some of the unique predicates introduced by replication can occur multiple times in the new rules added by replication (as illustrated in the following example).

EXAMPLE 5.2 Let a predicate p be defined by the following rules:

Algorithm 5.1 (RLCS/MST)

Input: A nonrecursive program P.
$\mathsf{IPG}(P)$, the information passing graph of program P.
Output: A nonrecursive program P' equivalent to program P s.t. MST(P') is nonrecursive.
Method

Let $\mathcal{C}$= the set of simple cycles in $\mathsf{IPG}(P)$;
if ($\mathcal{C} = \emptyset$) % $\mathsf{IPG}(P)$ *does not have cycles*
  Output Program P; % *We are done*
else % *Program MP is recursive*
  % Remove cycles in program MP
  while (there exists a cycle in set $\mathcal{C}$) % *Remove cycles one by one*
  {
   Choose a cycle C that has the least CSN amongst all cycles in set $\mathcal{C}$;
   Choose the smallest flex predicate p in cycle C;
   Let L_o be the label on the outedge from node p;
   Let L_i be the label on the inedge to node ip;
   For every rule $r \in (L_o - L_i) \cup (L_i - L_o)$ 1
     Replace predicate p by a new unique predicate p' in rule r.
     define_pred(p, p');
     modify($p, r, \mathcal{C}$); % modify *updates the set of all cycles* $\mathcal{C}$
  }

$\diamond$

p :- q & e.
p :- q & f.

e and f are edb predicates, and q is an idb predicate. If define_pred(p, p') is called, we get the following rules for predicate p':

p' :- q' & e.
p' :- q' & f.

Predicate q' occurs twice in the new rules thus created. define_pred(q, q') will be called. □

Therefore each application of define_pred produces at most one new predicate for every predicate in the old program.

Procedure modify alters the set of cycles $\mathcal{C}$ to capture the cycles in the IPG of the program P after predicate p has been replicated. Even though replicating a predicate using procedure define-pred can potentially introduce new cycles in the IPG, all the cycles are such that they are already "represented" in the set $\mathcal{C}$. That is, each new cycle c^{ν} introduced by replication is such that it is broken when some old cycle c^{ω} is broken. So no extra work needs to be done to break the new cycles. Phase 1 of procedure modify maintains those cycles where the replicated predicate p' and its information node ip' both participate, but where p' is not flex. We prove,

Algorithm 5.2 (define_pred)

Input: Undefined, new predicate p'.
Existing predicate p.
Output: Program P' such that p' is defined by same rules as p.
Method
Let R_p be the set of rules defining predicate p.
For every rule $r_p \in R_p$ **do**
Create a rule r'_p as follows:
(r'_p): s' :– $\mathsf{body}(r_p)$.
Let R'_p be the set of rules defining p' thus created.
For each derived predicate q occurring in R'_p
Replace q by a new unique predicate q';
define_pred(q, q'); *% i.e. replicate every derived subgoal in the new rule.*
Add the set of rules $R_{p'}$ to program P.
$\diamond$

in Section 5.3, that these are the only cycles that include any of the predicates introduced by replication. No other cycles use the replicated predicates. Phase 2 of procedure modify eliminates the edges due to rule r where ip is no longer passing information (or information is no longer being passed into p) because p has been replaced by p' in rule r. Elimination of these edges will break the cycles where node p was flex.

5.3 Proving Correctness of RLCS/MST

We first prove that the new nodes introduced by an application of define_pred are all non-flex (Lemma 5.1), and therefore do not need to be replicated for breaking cycles. We then go on to prove that the new nodes introduced by replication can never become flex (Theorem 5.1).

We then prove that nodes that can never be flex, need not be explicitly represented in cycles. That is, they can be short circuited and replaced by edges (Lemmas 5.4, 5.5, and 5.6). Because all the new nodes introduced by replication never become flex, they need not be represented in the cycles as long as their information passing behavior is correctly captured by appropriate edges. This guarantee allows us to avoid rebuilding the IPG when new predicates are introduced in the program by procedure define_pred. The above facts are sufficient to prove that the procedure modify is correct and retains all the information needed to break all the cycles in IPG(P) even though none of the new predicates introduced by replication are included in the cycles.

In order to prove correctness of Algorithm 5.1, we show that Algorithm 5.1 monotonically reduces a measure of cyclicity in the IPG. Therefore, the algorithm converges to an acyclic state of the IPG, which corresponds to guaranteeing a nonrecursive magic-sets transformation of the resulting program.

Definition 5.1 Primed predicates and rules: Let subgoal p in rule r be chosen for replication by RLCS/MST (Algorithm 5.1), following which the procedure define_pred (p, p')

Algorithm 5.3 (modify)

Input: p, the predicate to be replicated.
r, the rule in which p is to be replicated.
$\mathcal{C}$, the set of cycles in $\mathsf{IPG}(P)$.
Output: Updated $\mathcal{C}$.
Method
For every cycle $C \in \mathcal{C}$
% Phase 1
If C has both nodes p and ip
If $[r \in \mathsf{label}(\mathsf{outedge}(p))]$ **and** $[r \in \mathsf{label}(\mathsf{inedge}(p))]$ **then**
For the edge $s \Rightarrow ip$ in C **do**
If edge $p \Rightarrow t$ is in C **then**
Add edge $s \Rightarrow t$ to C.
If edge $p \rightarrow t$ is in C **then**
Add edge $s \rightarrow t$ to C.
% If the edge already exists then add r to its label.
% These new edges capture the intuition that the replicated predicate p' is non-flex in all
% cycles and can therefore be circumvented without affecting the information needed for
% breaking cycles (Lemma 5.2)
% Phase 2
For every $\mathsf{outedge}$ e_o from p **do**
Remove r from the label of e_o.
If $\mathsf{label}(e_o)$ is empty, **then** remove edge e_o.
For every inedge e_i into ip **do**
Remove r from the label of e_i.
If $\mathsf{label}(e_i)$ is empty, **then** remove edge e_i.
$\diamond$

is executed. Predicate p', and any other new predicate introduced by $\mathsf{define\_pred}$ will be called a *primed predicate*; the rules defining primed predicates will be called *primed rules*. If q' is a primed predicate, nodes q' and iq' in the IPGwill be called primed nodes.

A predicate that is not primed will be called a *nonprimed predicate*; its rules will be called *nonprimed rules*. If q is a nonprimed predicate, nodes q and iq in the IPGwill be called nonprimed nodes. $\square$

Lemma 5.1 *Let p be a subgoal to be replicated in rule r in program P. Let p' be the unique predicate that replaces p in r and let the resulting program be P'. Let $R_{p'}$ be the set of rules that define the predicate p' and all the new derived predicates introduced while defining p'. Let the set of new derived predicates occurring in $R_{p'}$ be Q (Clearly, $p' \in Q$).*

Then, none of the predicates in the set Q are flex with respect to any cycle. $\square$

Proof: Note that each primed rule r' in program P', is obtained by renaming of a non-primed rule r in P.

The predicate p' itself cannot be flex because it occurs exactly once in the program (Theorem 3.5). Let us consider the descendents of p', all of which are (new) primed predicates. Let q' be one such new predicate. Let C' be a cycle in $\mathsf{IPG}(P')$ such that q' is flex in C'. We show that a contradiction arises if this is the case.

Let us define a mapping ψ that maps a primed predicate to its corresponding non-primed predicate, *i.e.* ψ maps a replicated predicate p' to the predicate p. ψ can be extended, in a natural manner, to map a rule defining p' to the corresponding non-primed rule defining p, and then to map edges in $\mathsf{IPG}(P')$ to edges in $\mathsf{IPG}(P)$. Thus, for every edge e in the cycle $C' \in \mathsf{IPG}(P')$, there is a corresponding edge $\psi(e)$ in $\mathsf{IPG}(P)$. Therefore $\mathsf{IPG}(P)$ has a cycle C such that C is obtained from C' by the mapping ψ. Let q' be flex in C', say because the outedge from q' has an element r' that does not occur in the inedge to iq'. Then, in the cycle $C = \psi(C')$, the outedge from q has an element $r = \psi(r')$ in its label. However, q cannot be flex in cycle C because if q were flex then p would not be replicated in the last step of RLCS/MST (because q is a descendant of p (and hence smaller than p), and only the smallest flex predicate is replicated in each step). Therefore, the label on the inedge to iq in cycle C contains the element r. Because r' is the same as r except for renaming each predicate to its primed-version, the label on the inedge to iq' contains r', contradicting our assumption that q' is flex because of r'.

A similar argument can be made for the case if q' is flex because r' occurs on the inedge to iq' but not on the outedge from q'. $\square$

If the replication of flex predicates in the RLCS/MST algorithm was done in an arbitrary order (and not in increasing order of cycle stratum numbers), then the above theorem would not hold. This is so because then the new predicates created could also be flex in some cycle in $\mathsf{IPG}(P')$.

We now prove that the new predicates can *never* become flex. In order to prove that, we need Lemmas 5.2 and 5.3.

Lemma 5.2 *Given a program P, let subgoal p in rule r be chosen for replication by RLCS/MST (Algorithm 5.1). Let p' be the unique predicate that replaces p in r, and let the new resulting program be P'. The predicates in P are referred to as the non-primed predicates and the new predicates in P' as the primed predicates. Then, in $\mathsf{IPG}(P')$, every path from a non-primed node to a primed node includes the node ip'.* $\square$

Proof: The proof is by induction on the length of paths in $\mathsf{IPG}(P')$ from a non-primed node to a primed node.

<u>Path length $= 1$:</u> Note that p' is the only primed predicate that occurs as a subgoal in a non-primed rule. All other primed predicates occur only in primed rules. Therefore, the only paths of length 1 from, a non-primed node can be to p' or ip'. Lets first consider p'. All inedges to p' are **D** edges and these can only be from its descendants. All descendants of p' are primed. Hence there can be no paths of length 1 from non-primed nodes to p'. Hence, ip' is the only primed node that a non-primed node can reach by a path of length 1.

<u>Path length $= k + 1$:</u> We exhaustively consider all the possible types of the last edge in the path from a nonprimed node to a primed node. If the last node in the path is ip', then the inductive statement holds trivially. Let's consider the cases where the last node in the path is some primed node of the form q', or iq', where $iq' \neq ip'$. We will show that the second last node in this path is also primed. By the inductive hypothesis, the path to the second last node is of length k and therefore includes ip'. Therefore, the path to q' or iq' will also include ip':

- $s \rightarrow q'$: *i.e.* s is a subgoal in the rule defining q'. If q' is primed, then all the derived subgoals occurring in a rule defining q will also be primed; so s must be primed.
- $iq' \Rightarrow q'$: If q' is primed, then so is iq'.
- $s \Rightarrow iq'$, $iq' \neq ip'$: s has to be a subgoal in the same rule as q' for this edge to be in the IPG. If iq' is primed, then so is s).
- $is \Rightarrow iq'$, $iq' \neq ip'$: s has to be the head of a rule in which q' occurs as a subgoal. For q' to be primed, s also has to be primed (because all primed predicates other than p' occur as subgoals in primed rules).

□

Lemma 5.3 *Given a program P, let subgoal p in rule r be chosen for replication by RLCS/MST (Algorithm 5.1). Let p' be the unique predicate that replaces p in r, and let the new resulting program be P'. The predicates in P are referred to as the non-primed predicates and the new predicates in P' as the primed predicates. In* IPG(P')*, every path from a primed node to a nonprimed node includes the node p'.* □

Proof: Similar to the proof for Lemma 5.2. □

Before we prove that the new predicates can never be flex, we give the intuition for how a non-flex node can become flex. A non-flex node p in a cycle can become flex if the label on the inedge to the information node ip loses an element resulting in the outedge from p having an element that is not in the label of the inedge to ip. For this to happen, the node preceding ip needs to be replicated (and should therefore be flex). Alternatively, the node reachable from p could be replicated resulting in the outedge from p losing an element from its label and making p flex. This is illustrated in Example 5.4.

Theorem 5.1 *Given a program P, let subgoal p in rule r be chosen for replication by RLCS/MST (Algorithm 5.1), let p' be the unique predicate that replaces p in r, and let the set of new derived predicates introduced by* define_pred*($p.p'$) be Q. Then. a predicate in set Q cannot become flex with respect to any cycle during subsequent iterations of the RLCS/MST algorithm.* □

Proof: By Lemma 5.1, $q' \in Q$ can never be flex to start with. By Lemmas 5.2 and 5.3, all cycles that have node q' must also have node p'. By induction on the distance of q' from p', we prove that as the algorithm progresses, q' can never become flex because of a change to the label on the outedge from q'.

Distance from p' is 1: q' can never become flex because p' is itself not flex and will therefore never be replicated.

Distance from p' is $k+1$: Let the node connected to q' be s (or is). By the argument of the proof of Lemma 5.2, s is itself primed. In addition, s (is) is at a distance $\leq k$ from p' and will therefore never be replicated. Therefore, q' will never become flex.

A similar argument can be made to prove that q' can never become flex because of a change to the label on the inedge into iq'. □

We now argue that the procedure modify retains all the information necessary for making the IPG acyclic *i.e.* all the information about the flex nodes in the cycles.

Lemma 5.4 *Let node p be guaranteed to never be flex with respect to a cycle C. p can be ignored while choosing a node to replicate in order to break C.* $\Box$

Proof: Non-flex nodes are never replicated by Algorithm 5.1 and can therefore be ignored. $\Box$

Lemma 5.5 *Let node p be guaranteed to never be flex with respect to any cycle in the* IPG *of a program. Let C be a cycle that involves p. Let ip and the edge $ip \Rightarrow p$ also occur in C. Let the inedge to ip be $s \Rightarrow ip$ with label L. Let the outedge from p be $p \Rightarrow t$ (or $p \rightarrow t$). Eliminating nodes ip and p from the cycle by edge $s \Rightarrow t$ (or $s \rightarrow t$), with label L, does not affect subsequent decisions regarding the nodes to be replicated.* $\Box$

Proof: Because p is not flex, the label on the outedge from p is also L. Because p is guaranteed to never be replicated (p is non-flex with respect to all cycles), the inedge to t and the outedge from s will always have the label L. Therefore eliminating p and ip does not affect the behavior of either t or s. $\Box$

Note that in Lemma 5.5 we needed p to be non-flex with respect to all cycles in the IPG and not just with respect to cycle C. This is because replicating p in order to break some other cycle can render p flex with respect to cycle C.

Lemma 5.6 *Let node p be guaranteed to never be flex with respect to any cycle C in the* IPG*. Consider a cycle C such that both p and ip occur in C. If all the nodes on the path from ip to p are also guaranteed to never be flex with respect to any cycle in the* IPG*, then the path $ip \rightarrow\!\!\!\rightarrow p$ can be replaced by a single edge.* $\Box$

Proof: Similar to the proof of Lemma 5.5. $\Box$

Lemma 5.7 *Given a program P, let subgoal p in rule r be chosen for replication by RLCS/MST. Let p' be the unique predicate that replaces p in r, and let the new resulting program be P'. All cycles in* IPG*(P') either have both nodes p' and ip' or neither.* $\Box$

Proof: The predicates in P are referred to as the non-primed predicates and the new predicates in P' as the primed predicates. Say cycle C has node p' but not node ip'. C has to have at least one non-primed node that is flex. If not, then all nodes in C would be primed and non-flex (Theorem 5.1) which would contradict Theorem 3.2. Let this non-prime flex node be x. ix is also in C because x is flex. For ix and p to be in a cycle, there is a path from ix to p'. By Lemma 5.2, path $ix \rightarrow\!\!\!\rightarrow p'$ has to involve ip'.

Similarly, if cycle C has node ip' but not node p', we consider the path from ip' to the non-primed node x to get a contradiction. $\Box$

Theorem 5.2 *Given a program P, let subgoal p in rule r be chosen for replication by RLCS/MST. Let p' be the unique predicate that replaces p in r, and let the new resulting program be P'. Phase 1 of procedure* modify *alters the set of cycles $\mathcal{C}$ to give a new set $\mathcal{C}'$ such that all the cycles of* IPG(P') *are guaranteed to break when the cycles in $\mathcal{C}$ are broken.* $\Box$

Proof: By Theorem 5.1, all primed nodes introduced by replication are guaranteed to be non-flex. Therefore, they can be short-circuited in the IPG without losing any information (Lemma 5.6). Phase 1 does precisely this short-circuiting, *i.e.* alters $\mathcal{C}$ such that every cycle that has both p' and ip' is short-circuited and added to the new $\mathcal{C}$. □

By Lemma 5.7, there are no cycles that have only p' or only ip'. Phase 2 of procedure modify eliminates those edges that do not participate in any cycles. Phase 2 therefore eliminates all those cycles that are broken by replication, but not those that are introduced by replication (and introduced in Phase 1).

We now define a measure of cyclicity for an IPG in order to prove that Algorithm 5.1 results in a program that has an acyclic magic-sets transformation.

Definition 5.2 flexibility coefficient: Let C be a cycle in IPG(P). Let the set of predicate nodes that occur in C be pred(C). Let flex_pred(C) be those predicates in pred(C) that *can* be flex, *i.e.* flex_pred(C) is pred(C) without those predicate nodes that are guaranteed not to be flex. flex_pred(P) is the union of the sets flex_pred(C) over all simple cycles in IPG(P). The flexibility coefficient of IPG(P) is defined to be the sum of the cardinalities of (1) the labels on the outedges of the members of flex_pred(P), and (2) the labels on the inedges of the information nodes of the members of flex_pred(P). □

Note that the flexibility coefficient of an IPG with no cycles is 0. And conversely, if the flexibility of an IPG is 0, then the IPG is acyclic.

Lemma 5.8 *Let p be a subgoal to be replicated in rule r in program P. Let p' be the unique predicate that replaces p in r and let the new resulting program be P'. The flexibility coefficient of* IPG*(P') is strictly less than the flexibility coefficient of* IPG*(P).* □

Proof: By Theorem 5.1, none of the new nodes introduced by replication will be flex. Therefore the set flex_pred(P) does not grow by replication. We now argue that the flexibility coefficient of IPG(P') is actually reduced. Procedure modify removes $\{r\}$ from the labels on all outedges and inedges from p and ip. The cardinality of the label on the outedge from p and the inedge into ip necessarily reduces. □

Note that the transformation algorithm iterates as long as there are any cycles in the IPG of the transformed program. In addition, by Lemma 5.1, the flexibility coefficient of the transformed program is strictly lower than the flexibility coefficient of the program before the iteration. Therefore, the algorithm terminates and does so yielding a program whose IPG has no cycles.

To prove that the resulting program is equivalent to the original program note that replication preserves equivalence and replication is the only transformation applied to the program.

Theorem 5.3 *Algorithm 5.1 transforms a program P into program p' such that* IPG(P') *is acyclic.* □

The proof of the above theorem is a combination of the other lemmas in this section.

5.4 Optimizations

Algorithm 5.1 can be optimized by reducing the number of instances of a flex predicate that are replicated. In Step 1, the RLCS/MST algorithm, as written, replicates occurrences of the predicate p that occur in the rules in $(L_o - L_i) \cup (L_i - L_o)$. Each element in the label L_i represents a way of passing information into the node ip. Similarly, each element in the label L_o represents a way of passing information out of node p. Replicating an instance of p that occurs in a rule in L_o, corresponds to replacing the node p by the node p', thereby breaking a path from ip to p. Similarly, replicating an instance of p that occurs in a rule in L_i, corresponds to replacing the node ip by a node ip'. Breaking all paths from ip to p also breaks the cycle that use this path.

Therefore, if the sets L_o and L_i are disjoint, then replicating the occurrences of p in any one of the two sets $(L_o - L_i)$ or $(L_i - L_o)$ is sufficient to break all paths from ip to p, and the RLCS/MST algorithm can be optimized for the above. That is, the occurences of p in just one of the two sets $(L_o - L_i)$ and $(L_i - L_o)$ need be replicated to break the cycle under consideration. However, when sets L_o and L_i are not disjoint, this optimization cannot be made. The reason is that in this case, replicating p does not break the cycle, instead only reduces the number of alternative paths from ip to p. In addition, not replicating the occurences of p in either $(L_o - L_i)$ or $(L_i - L_o)$ keeps p flex (Definition 3.4) causing p to be picked up by the algorithm to be replicated again. Hence, the optimization is likely to be ineffective when L_o and L_i are not disjoint. A useful heuristic to follow while selecting nodes to replicate is to pick those nodes where L_o and L_i are disjoint.

A second optimization is possible in Phase 2 of Algorithm modify. Currently Phase 2 is written to eliminate label r from all remaining outedges from predicate node p, and all remaining inedges into information node ip. However, only those edges that are part of some cycle in set $\mathcal{C}$ need be eliminated.

5.5 Examples

EXAMPLE 5.3 Consider Algorithm 5.1 applied to the program in Example 1.1.

(Q): ?- $t(0)$.

$(P1)$: $t(X)$:- $s(X,Y)$ & $p(Y)$.

$(P2)$: $s(X,Y)$:- $p(X)$ & $q(X,Y)$.

$(P3)$: $p(X)$:- $e_1(X,Y)$ & $e_2(Y)$.

$(P4)$: $q(X,Y)$:- $e_3(X,Z)$ & $e_4(Z,Y)$.

The cycles are as illustrated in Example 3.2. Both cycles have the same CSN so we pick one arbitrarily. Say we pick cycle $C1$. The only flex node in $C1$ is p. L_o is $\{2\}$ *i.e.* rule $P2$, and L_i is $\{1\}$. We replicate the occurrence of p in $L_0 - L_i$ *i.e.* rule 2 only (using the optimization discussed above). This results in breaking the edge $p \overset{\{2\}}{\Rightarrow} iq$ in cycle $C1$.
Procedure define_pred creates a predicate $p1$ that is defined exactly as p resulting in the following program:

Cycle Identifier	Elements in Order of Traversal	Flex node(s)	CSN
$C1$	$p \overset{\{2\}}{\Rightarrow} iq \Rightarrow q \overset{\{2\}}{\rightarrow} s \overset{\{1\}}{\Rightarrow} ip \Rightarrow p$	p	1
$C2$	$p \overset{\{2\}}{\Rightarrow} s \overset{\{1\}}{\Rightarrow} ip \Rightarrow p$	p	1

(Q): ?- $t(0)$.

$(P1)$: $t(X)$:- $s(X,Y)$ & $p(Y)$.

$(P2)$: $s(X,Y)$:- $p1(X)$ & $q(X,Y)$.

$(P3)$: $p(X)$:- $e_1(X,Y)$ & $e_2(Y)$.

$(P3')$: $p1(X)$:- $e_1(X,Y)$ & $e_2(Y)$.

$(P4)$: $q(X,Y)$:- $e_3(X,Z)$ & $e_4(Z,Y)$.

No new predicates are created while defining predicate $p1$ because both the predicates e_1 and e_2 used in defining p (and therefore $p1$) are base predicates. If either of them was a derived predicate, then new predicates would have been created. For instance, if e_1 was a derived predicate, then we would have created a new predicate e_{11} that would have been defined by the same rules as those defining e_1.

Procedure modify does not find any cycle that has both nodes ip and p with the element 2 in the labels of both outedge(p) and inedge(ip). Therefore, Phase 1 does not do anything. In Phase 2 the edges $p \overset{\{2\}}{\Rightarrow} iq$ and $p \overset{\{2\}}{\Rightarrow} s$ both get broken. This happens because both edges lose the element 2 from their labels, rendering both labels empty and breaking the corresponding edges. Cycle $C2$ gets broken as a result, terminating the algorithm because no cycles remain. □

EXAMPLE 5.4 Consider the program $\mathcal{T}$:

(Q): ?- $t(0)$.

$(T1)$: $t(X)$:- $p(X)$ & $q(X)$ & $s(X)$.

$(T2)$: $t(X)$:- $q(X)$ & $s(X)$ & $p(X)$.

$(T3)$: $t(X)$:- $s(X)$ & $p(X)$ & $q(X)$.

We omit the rules that define the predicates p, q, and s. A part of the information passing graph for program $\mathcal{T}$ is shown in Figure 4. We have chosen not to show node t in the IPG because t is not part of any cycle in the IPG. There is only one cycle in the IPG and the CSN of the cycle is 1 (assuming that p, q, s are all defined using base predicates).

All three nodes p, q, and s are flex. We pick one of the three nodes arbitrarily, say p. The label on inedge(ip) is $\{2, 3\}$ and label on outedge(p) is $\{1, 3\}$. Because the two labels are not disjoint, we replicate the occurrences of p in every rule in $(L_o - L_i) \cup (L_i - L_o)$ *i.e.* in rules $\{1, 2\}$. Note that the two occurrences get replaced by different new predicates $p1$, and $p2$. The replication results in the program:

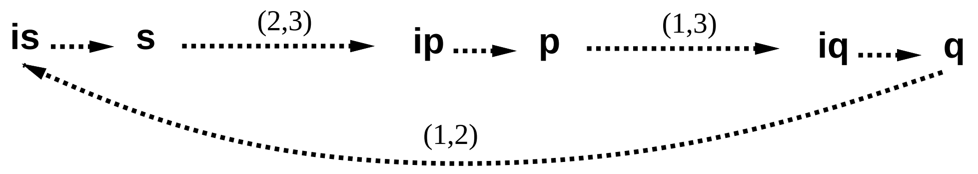

Figure 4: Information Passing Graph of program T

(Q): ?- $t(0)$.

$(T1)$: $t(X)$:– $p1(X)$ & $q(X)$ & $s(X)$.

$(T2)$: $t(X)$:– $q(X)$ & $s(X)$ & $p2(X)$.

$(T3)$: $t(X)$:– $s(X)$ & $p(X)$ & $q(X)$.

The corresponding IPG is shown in Figure 5:

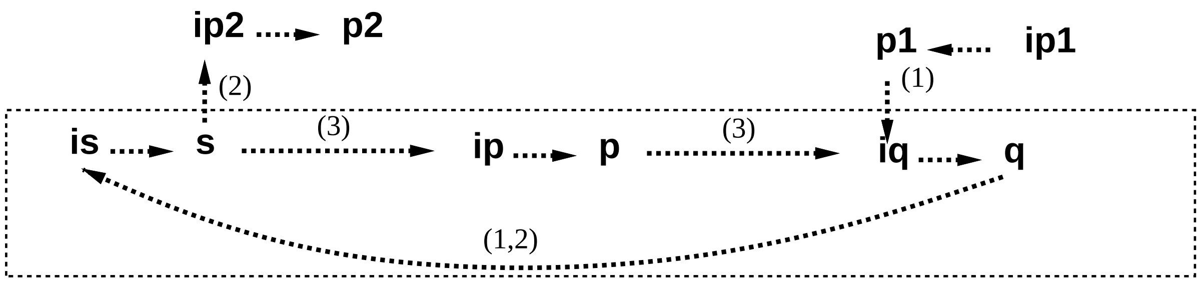

Figure 5: IPG of Program T after replicating two instances of p

In the IPG of the transformed program, p is non-flex. Procedure modify changes the original cycle to the part of the IPG that is in the box. Nodes q and s are both flex in this cycle.

Say we pick up node s for replication. Note that the label on inedge(is) is disjoint from the label on outedge(s). We pick the smaller of the two labels, namely $\{3\}$, and replace the occurrence of s in rule $T3$ by a new predicate $s1$. The resulting program is as follows:

(Q): ?- $t(0)$.

$(T1)$: $t(X)$:– $p1(X)$ & $q(X)$ & $s(X)$.

$(T2)$: $t(X)$:– $q(X)$ & $s(X)$ & $p2(X)$.

$(T3)$: $t(X)$:– $s1(X)$ & $p(X)$ & $q(X)$.

The resulting program has an acyclic IPG and consequently the magic-sets transformation of this program is nonrecursive.
□

Instead of replicating predicates in Algorithm 5.1, the marked predicate could be completely *unfolded*. The correctness of the resulting algorithm can easily be proven from the lemmas proven in this section.

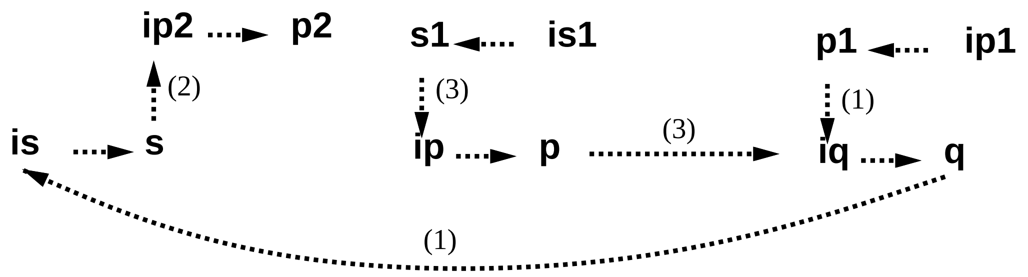

Figure 6: IPG of Program T after replicating s

6 Covered Subgoal Elimination

In this section we describe a general optimization technique that can be used to eliminate (recursive) subgoals from a datalog program.

EXAMPLE 6.1 (Motivation): Consider the program S:

$(S1)$: $s(X,Y)$:- $e_1(B,Y,Z)$ & $e_2(Y)$ & $q(Y,Z)$ & $p(X,Z)$.

$(S2)$: $q(Y,Z)$:- $e_1(A,Y,Z)$.
$(S3)$: $q(Y,Z)$:- $s(X,Y)$ & $p(Y,Z)$.

If a full sips is used in rule $S1$, the subgoals $e_1(B,Y,Z)$ and $e_2(Y)$ bind variable Y and Z before subgoal $q(Y,Z)$ is evaluated. The subgoal $q(Y,Z)$ then provides an existential test for the assigned (Y,Z) values. However, rule $S2$ defines $q(Y,Z)$ to be true whenever $e_1(B,Y,Z)$ is true, making the existential test redundant. The subgoal $q(Y,Z)$ can thus be eliminated from rule $S1$ to derive an equivalent program T:

$(T1)$: $s(X,Y)$:- $e_1(B,Y,Z)$ & $e_2(Y)$ & $p(X,Z)$.

$(T2)$: $q(Y,Z)$:- $e_1(A,Y,Z)$.
$(T3)$: $q(Y,Z)$:- $s(X,Y)$ & $p(Y,Z)$.

Note that eliminating subgoal $q(Y,Z)$ transformed a recursive program S into an equivalent nonrecursive program T. $\square$

We now describe a general algorithm to eliminate redundant subgoals as in the example above. The algorithm uses the idea of containment mappings [Ull89]. However, we note that the conjunctive query containment techniques discussed in [Ull89] cannot eliminate subgoal $q(Y,Z)$ from rule $S1$ in the example above.

Definition 6.1 Cover: Consider the rule r:

(r): s :- g_1 & ... & g_k & q & g_{k+1} & ... & g_n.

with $q, g_1, \ldots, g_n$ as subgoals. Let the function VAR return the variables that occur as arguments in a given set of subgoals. Let $\mathcal{P} \subseteq \{g_1, \ldots, g_n\}$ be the set of subgoals that share a variable with q. A set of subgoals $\mathcal{C} \subseteq \mathcal{P}$ is said to be a *cover* for subgoal q if $\mathsf{VAR}(q) \subseteq \mathsf{VAR}(\mathcal{C})$. Subgoal q is said to be *covered* by set $\mathcal{C}$. $\square$

EXAMPLE 6.2 Consider the rule $S1$ from Example 6.1. The set $\{e_1(B,Y,Z)\}$ is a cover for the subgoal $q(Y,Z)$. The set $\{e_1(B,Y,Z), e_2(Y)\}$ is also a cover for subgoal $q(Y,Z)$, but it is not minimal.
$\Box$

The subgoals in a cover determine a set of bindings for the variables in the covered subgoal. If we can guarantee that the covered subgoal will be true for each of these bindings, then the covered subgoal is redundant.

Definition 6.2 Covered Subgoal Elimination: Given a program with the following pair of rules:

$(r1)$: $p(t_1, \ldots, t_n)$:- S_1 & $q(u_1, \ldots, u_m)$ & S_2.

$(r2)$: $q(v_1, \ldots, v_m)$:- S_3.

where S_1, S_2, and S_3 are conjunctions of subgoals, consider the following two conditions:

1. There exists a cover S_0 for the subgoal $q(u_1, \ldots, u_m)$ in rule $r1$.
2. Let rule $r3$ be defined as follows:

   $(r3)$: $q(u_1, \ldots, u_m)$:- S_1 & S_2.

   Then, there is a containment mapping [Ull89] from rule $r2$ to rule $r3$. That is, if rules $r2$ and $r3$ are viewed as conjunctive queries, the conjunctive query $r3$ is contained in the conjunctive query $r2$.

If Conditions 1 and 2 are satisfied, then subgoal $q(u_1, \ldots, u_m)$ can be eliminated from rule $r1$. $\Box$

EXAMPLE 6.3 Consider program S introduced in Example 6.1. The set $\{e_1(B,Y,Z)\}$ is a cover for the subgoal $q(Y,Z)$. Also, there exists a containment mapping from rule

$q(Y,Z)$:- $e_1(A,Y,Z)$.

to the rule

$q(Y,Z)$:- $e_1(B,Y,Z)$ & $e_2(Y)$ & $p(X,Z)$.

The containment mapping is $\{Z \rightarrow Z, Y \rightarrow Y, A \rightarrow B\}$. Subgoal $q(Y,Z)$ is thus redundant, and can be eliminated from rule $S1$. $\Box$

Theorem 6.1 *Let CP be the program obtained by eliminating a redundant subgoal from a given program P using covered subgoal elimination. Programs P and CP are equivalent.* $\Box$

Proof: In the program *CP*, rule $r1$ is replaced by rule $r1'$ after covered subgoal elimination:

$(r1)$: $p(v_1, \ldots, v_n)$:- S_1 & $q(u_1, \ldots, u_m)$ & S_2.

$(r1')$: $p(v_1, \ldots, v_n)$:- S_1 & S_2.

To prove that program P is equivalent to *CP*, it is enough to prove that rules $r1'$ and $r1$ derive the same facts.

$r1 \subseteq r1'$: Eliminating a subgoal from the body of a rule can only increase the facts derived by the rule. Rule $r1'$ is obtained by eliminating subgoal q from rule $r1$ and will therefore derive all facts derived by rule $r1$.

$r1' \subseteq r1$: The proof is by contradiction. Say $r1'$ derives a fact that $r1$ does not. The only difference between rules $r1$ and $r1'$ is in the subgoal q. In rule $r1$, subgoal $q(u_1, \ldots, u_m)$ eliminates some of the bindings for variables $u_1, \ldots, u_m$ assigned by the cover of q. However consider rule $r2$ from Definition 6.2. Condition 2 in Definition 6.2 implies the existence of a symbol mapping from the body of rule $r2$ to the body of rule $r1'$. Therefore the values computed by rule $r2$ are a superset of the tuples permitted by the subgoals in rule $r1'$. Hence subgoal q cannot eliminate any variable bindings in rule $r1$. $\Box$

Covered subgoal elimination can be used in programs where duplicates are important. Using duplicate semantics for programs with `set` and `multiset` predicates (such as SQL programs) as defined in [MPR90], we have the following result:

Theorem 6.2 *Let CP be the program obtained by eliminating redundant subgoals for* `set` *predicates from a given program P using covered subgoal elimination. Programs P and CP are duplicate equivalent.* $\Box$

Proof: Let p be a predicate eliminated by covered subgoal elimination from a rule in program P. Consider a proof tree for some fact $f(s)$, $\mathsf{T}(f(s))$, that has a node corresponding to a fact for predicate p, $f(p)$. Multiple proof trees for fact $f(p)$ result in multiple proof trees for $f(s)$ keeping everything else in the proof tree for $f(s)$ the same. If p is a `set` predicate in the program, then only one proof tree for $f(p)$ will be considered. If CSE is then used to eliminate the subtree $\mathsf{T}(f(p))$, the remaining proof tree for $f(s)$ is not affected. However, if p were not a `set` predicate, eliminating $f(p)$ would result in a reduction in the total number of proof trees because all proof trees for $f(s)$ that differed only in $\mathsf{T}(f(p))$ will collapse into one. $\Box$

7 Conclusions

We have solved an important problem: how to use the magic-sets transformation in database systems that do not support recursion.

We have developed magic-sets transformation algorithms that guarantee a nonrecursive output when given a nonrecursive program as input. These algorithms are useful because they make the efficient magic-sets transformation (MST) usable by current systems that do not support recursion. The performance improvements obtainable by MST can now be enjoyed in nonrecursive systems.

We have discussed one nonrecursive magic-sets transformation algorithm, RLCS/MST, in detail. The unfold version of the algorithm can be built very simply by unfolding predicates instead of replicating them. The algorithm work by selective replication of common subexpressions. Both algorithms can preserve duplicate semantics [MPR90], and can therefore be used in SQL systems. However, unfolding is not always possible in SQL queries (as when doing grouping), and RLCS/MST may be be the only applicable algorithm on such queries. Both the algorithms work with adorned programs; we presented them using unadorned programs for simplicity.

There are a few problems that need further work. In what order should one break the cycles with the least CSN in the information passing graph? Within a cycle, how should one select common subexpressions to be replicated so as to minimize the amount of replication? What is the performance penalty due to replication? RLCS/MST can still replicate common subexpressions unnecessarily, as the following example shows. We would like to develop an algorithm that replicates a common subexpression only if the replication is necessary to avoid recursion.

EXAMPLE 7.1 Consider the program N:

(1): $t(X) :- s(X) \ \& \ r(X,Y)$.
(2): $u(X) :- r(X,Y) \ \& \ p(Y,Z) \ \& \ q(Z,A)$.
(3): $v(X) :- q(X,Y) \ \& \ s(Y)$.
(4): $w(X) :- q(X,Y) \ \& \ p(Y,Z)$.

The IPG fpr the program is as follows (we omit those parts of the IPG that are not critical for illustrating the non-optimality of RLCS/MST):

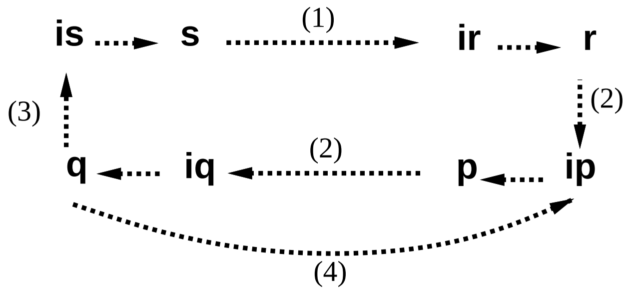

Figure 7: IPG of Program N

There are two cycles in the IPG. If the occurrence of q in rule (2) is replicated, both cycles are broken. Replication will replace node iq at the head of edge (2) with the newly introduced node iq'. There will be no cycle with both nodes q' and iq' and therefore modify (in Phase 1) will not retain either of the two cycles. However, if the algorithm first replicates predicate s in rule (1) (or r in rule (1)), then two predicates will have to be replicated. After the first replication, s (ir) will be replaced by s' (ir') and therefore, only one of the two cycles will be broken by modify (Phase 2). The other cycle will stay as it was before replication. □

A second contribution of this paper is the *covered subgoal elimination* (CSE) technique to optimize rule evaluation. CSE is related to conjunctive query minimization and minimization of unions of conjunctive queries ([Ull89], Chapter 14) in that they all use conjunctive query containment as a tool. CSE checks redundancy of subgoals by looking at other rules in the program. Redundant subgoals are then eliminated, and a recursive query may thus become nonrecursive.

With the performance advantage of the magic-sets transformation on nonrecursive queries well established [MFPR90], we expect this paper to have a real impact on how query optimization is done in commercial database systems.

8 Acknowledgements

We thank Hamid Pirahesh and Jeff Ullman for discussions, and Oded Shmueli for valuable comments on a draft of the paper.

References

[BPRM91] Isaac Balbin, G. S. Port, Kotagiri Ramamohanarao, and K. Meenakshi. Efficient Bottom-Up Computation of Queries on Stratified Databases. *Journal of Logic Programming*, December, 1991.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the Sixth Symposium on Principles of Database Systems (PODS)*, pages 269–283, San Diego, CA, March 1987. ACM SIGACT-SIGMOD-SIGART.

[GM92] Ashish Gupta and Inderpal S. Mumick. Magic-Sets Transformation in Non-Recursive Systems . In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, pages 354–367, 1992.

[ISO90] ISO_ANSI. Iso-ansi working draft: Database language sql2 and sql3; x3h2; iso/iec jtc1/sc21/wg3, 1990.

[KSS91] D.B. Kemp, P.J. Stuckey, and Divesh Srivastava. Magic-Sets and Bottom-Up Evaluation of Well-Founded Models. In *International Symposium on Logic Programming*, 1991.

[MFPR90] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In *Proceedings of ACM SIGMOD 1990 International Conference on Management of Data*, pages 247–258, Atlantic City, NJ, May 23-25 1990.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, pages 264–277, Brisbane, Australia, August 13-16 1990.

[Mum91] Inderpal Singh Mumick. *Query Optimization in Deductive and Relational Databases.* PhD thesis, Stanford University, Stanford, CA 94305, USA, December 1991.

[Nau86] Jeffrey F. Naughton. Data independent recursion in deductive databases. In *Proceedings of the Fifth Symposium on Principles of Database Systems (PODS)*, pages 267–279, Cambridge, MA, 1986.

[Ros90] K.R. Ross. Modular Stratification and Magic-Sets for Datalog Programs with Negation. In *Proceedings of the Ninth Symposium on Principles of Database Systems (PODS)*, pages 161–171, 1990.

[RSS90] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Rule Ordering in Bottom-Up Fixpoint Evaluation of Logic Programs. In *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, pages 359–371, Brisbane, Australia, 1990.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformations in logic programs. In S. A. Tarnlund, editor, *Proc. of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes 1 and 2*. Computer Science Press, 1989.