May 1986

Report No. STAN-CS-86-1114

Optimizing Function-Free Recursive inference Rules

by

Jeffrey F. Naughton

Department of Computer Science

Stanford University
Stanford, CA 94305

Optimizing Function-Free Recursive Inference Rules

Jeffrey F. Naughton*
Stanford University

May 1986

Abstract

Recursive inference rules arise in recursive definitions in logic programming systems and in database systems with recursive query languages. Let *D* be a recursive definition of a relation ***t***. We say that *D* is ***minimal*** if for any predicate ***p*** in a recursive rule in *D*, ***p*** must appear in a recursive rule in any definition of ***t***. We show that testing for minimality is in general undecidable. However, we do present an efficient algorithm for a useful class of recursive rules, and show how to use it to transform a recursive definition to a minimal recursive definition. Evaluating the optimized definition will avoid redundant computation without the overhead of caching intermediate results and run-time checking for duplicate goals.

1 Introduction

In recent years there has been increasing interest in extending relational database systems by adding query languages that allow relations to be defined recursively. Also, there is an increasing need to apply database techniques to logic programming systems so that they can efficiently handle large amounts of data. In both cases, recursive inference rules are a source of power and a source of inefficiency.

Function-free inference rules are an important sub-class of inference rules. They are particularly useful in "knowledge base" applications, which deal primarily with relationships between objects rather than with the structure of individual objects. There is currently a great deal of effort devoted to finding efficient evaluation algorithms for recursive function-free inference rules [2,3,6,8,15]. In this paper we consider a related issue: redundancy that arises from the specification of the rule, independent of the particular evaluation algorithm used.

*Work supported by NSF grant IST-84-12791 and a grant from the IBM Corporation.

The relations in our model can be divided into two groups. The extensional database, or ***EDB,*** is equivalent to a traditional relational database. The intensional database, or ***IDB,*** is a set of inference rules that define relations not stored explicitly in the EDB. These inference rules are function-free Horn clauses. (The terminology used here follows that of Reiter [12].) We will use Prolog notation to write these rules.

In this paper we consider only linear recursive rules, that is, rules in which the predicate in the head appears once in the body. We also require that there be no constants and no repeated variables in the rule head. A template for such a recursive rule is:

r_r : $t :- t, p_1, p_2, \ldots, p_n.$

The p_i need not be EDB predicates — the only restriction is that they cannot be mutually recursive with t. The p_i need not be distinct, but for clarity of exposition we will refer to p_i and p_j, $i \neq$ j, as "different predicates" rather than "different predicate occurrences." The recursively defined predicate t must also have some nonrecursive initialization. Unless noted otherwise, we will assume some generic initialization

r_e : $t :- t_0.$

where all the variables in t appear in t_0.

Informally, a recursive definition R is minimal if no predicate can be removed from any recursive rule in R. If R is not minimal, it contains some kind of redundancy. The most obvious form of redundancy in recursive rules arises because the body, when viewed as a relational expression, is redundant.

In the following rule, the second ***p*** predicate can be removed.

r_1: $t(X,Y) :- t(X,W), p(Y,W), e(W,Y), p(Y,Z).$

This kind of redundancy can be detected and removed by well-known techniques for optimizing expressions of relational algebra [1,4]. However, a recursive rule with a body that is minimal as a relational expression is not necessarily minimal.

Consider the following pair of rules, defining a relation ***buys(X,*** Y) of buyers and the products they buy.

r_2: *buys*(X, Y) :- *likes*(X, Y), *cheap*(Y).

r_3: *buys*(X, Y) :- *knows*(X, W), *buys*(W, Y), *cheap*(Y).

In this definition, the ***cheap*** predicate can be removed from r_3. We say that a predicate occurrence like ***cheap*** in r_3 is ***recursively redundant.***

Some very similar recursive rules are minimal. In the rules

r_4: *buys*(X, Y) :- *rich*(X), *likes*(X, Y).

r_5: *buys*(X, Y) :- *rich*(X), *knows*(X, W), *buys*(W, Y).

the ***rich*** predicate cannot be removed from r_5.

There is an essential difference between ***cheap*** in r_3 and ***rich*** in r_5. Any tuple (X, Y) in the relation ***buys*** of r_2 and r_3 depends on exactly one tuple of ***cheap*** for its proof. However, there is no a priori bound on the number of tuples of ***rich*** needed to prove that a tuple (X, Y) is in the relation ***buys*** of r_4 and r_5.

The full definitions of recursively redundant predicates and minimal recursive definitions follow:

Definition 1.1 Let r be a recursive rule in a set of rules defining a relation t, and let p appear in r. Then p is ***recursively redundant*** if there is some k such that no tuple of t depends on more than k tuples of p for its proof.

Definition 1.2 A recursive definition is minimal if it contains no recursively redundant predicates.

If every predicate in a recursive rule is recursively redundant, then the recursively defined relation can instead be defined by a finite number of nonrecursive rules. Thus an algorithm to find redundant predicates would also solve the "bounded recursion" problem: given a recursively defined relation, can it instead be defined by a first order expression?

Previous work by Ioannides [7], Minker and Nicolas [10], Naughton [11], and Sagiv [13], has solved the bounded recursion problem for various restricted classes of rules. Gaifman [5] has recently shown that the general case is undecidable. This implies that the "recursively redundant predicate" problem is also undecidable, that is, there is no algorithm which will find all recursively redundant predicates in an arbitrary recursive definition.

However, we are able to give a linear-time algorithm that, while always correct, is not necessarily complete. Furthermore, we identify a useful class of rules for which the algorithm is complete. The proof of the algorithm suggests a procedure to convert a set of rules containing a recursive rule to a new set containing an optimized recursive rule. In the simplest case (as in rules r_2 and r_3 above) the optimization merely removes predicates from the recursive rule; in general, it will modify the nonrecursive rules as well. For rules for which the algorithm is complete, this new recursive rule is guaranteed to be minimal.

2 Expansions and Redundancy

The expansion of an IDB predicate t is the set of all conjunctions of EDB predicates that can be generated by some sequence of rule applications to t. For recursive predicates, the expansion is infinite. There is a close connection between recursively redundant predicates and the minimality of the individual elements of the expansion.

2.1 Expansions of Recursive Definitions

Recall that in this paper, we are dealing with linear recursive rules without constants or repeated variables in the rule head. Procedure ExpandRule (Figure 1), enumerates the expansion of such definitions consisting of a recursive rule, r_r, and a nonrecursive rule, r_e. The output of ExpandRule is the expansion of the recursively defined predicate, represented by the infinite set S.

```
1)  Give all variables in rules subscript 0;
2)  ω := θ;
3)  CurString := t;
4)  while true do
6)      S := S ∪ {CurString with r_e applied};
7)      CurString := CurString with r_r applied;
8)      increment the subscripts of all variables in r_r and r_;
9)  endwhile;
```

Figure 1: Procedure ExpandRule

Throughout the procedure, the string-valued variable CurString will have exactly one occurrence of the recursive predicate *t*. To "apply" a rule *r* to CurString, replace that occurrence of *t* by the right side of *r*, after the substitutions required to unify it with the head of the rule. In the initialization, we subscript the variables in the rules so that no variable appears in both CurString and one of the rules. On each iteration, we increment the subscripts for the same reason.

Example 2.1 If e is the edge relation of a digraph, then the following rules define the transitive closure *t* of the graph.

r_r: $t(X, Y) :- e(X, Z), t(Z, Y).$

r_e: $t(X,Y) :- e(X,Y).$

Since e and *p* are identical in this case, we let e denote the occurrence of e in the recursive rule, and e' denote the occurrence in the nonrecursive rule. The first four strings in the set S are

$$e'(X,Y),$$
$$e(X,Z_0)e'(Z_0,Y),$$
$$e(X,Z_0)e(Z_0,Z_1)e'(Z_1,Y),$$
$$e(X,Z_0)e(Z_0,Z_1)e(Z_1,Z_2)e'(Z_2,Y).$$

∎

A string in an expansion may contain multiple occurrences of each predicate appearing in the recursive rule. We will use "predicate instance" to refer to occurrences of predicates in the strings of the expansion.

The strings in an expansion are conjunctive queries, a subset of relational expressions. If a variable *V* appears in the head of the rule, then *V* is a ***distinguished*** variable; otherwise,

it is ***nondistinguished.*** If $V_1, V_2, \ldots, V_i$ are the distinguished variables, and $W_1, W_2, \ldots, W_j$ the nondistinguished variables, then the relation specified by the string $p_1 p_2 \ldots p_n$ is

$$\{(V_1, V_2, \ldots, V_i) | (\exists W_1, W_2, \ldots, W_j)(p_1 \ p_2 \text{ A } \ldots \text{ A } p_n)\}$$

The recursively defined relation is the union of the relations for the strings in the expansion.

In the next section we will need to decide equivalences between conjunctive queries; to do this, we use techniques related to tableaux mappings, a tool developed by Aho et al. [1].

Definition 2.1 A mapping m from the variables of a string s_1 to the variables of a string s_2 is a ***containment mapping*** if distinguished variables map to themselves, and if $p(X_1, \ldots, X_n)$ appears in s_1, then $p(m(X_1), \ldots, m(X_n))$ appears in s_2.

The following lemma shows the similarity between this mapping and containment mappings for deciding the equivalence of tableaux.

Lemma 2.1 ***If a string s_1 maps to a string s_2, then the relation specified by s_2 is contained*** in ***the relation specified by s_1.***

Proof: The proof follows that of containment mappings for tableaux [1]. ∎

Another useful relationship between strings is ***isomorphism.***

Definition 2.2 Two strings are ***isomorphic*** if they are identical up to renaming of nondistinguished variables.

There are containment mappings in both directions between isomorphic strings. In any given set of rules, the number of predicates and distinguished variables is finite. Because of this, isomorphism is an equivalence relation that partitions all strings of a given length into a finite number of equivalence classes. Any two strings from the same equivalence class specify exactly the same relation.

Because there are a finite number of nonisomorphic instances of the recursive predicate t, -if we observe the instances of t in the variable *CurString* in Procedure ExpandRule, we must eventually see two that are isomorphic. If the number of iterations between these isomorphic instances is τ, there will be a constant σ such that, for all $m \geq 0$, the instances of t produced on iterations $\sigma + \tau + i$ and $\sigma + m\tau + i$ are isomorphic. The constant τ is called the ***period*** of the rule.

2.2 Recursively Redundant Predicates

Recall the definition of a recursively redundant predicate: Let r be a recursive rule in a set of rules defining a predicate t, and let p appear in r. Then p is ***recursively redundant*** if there is some k such that no tuple of t depends on more than k tuples of p for its proof. More formally, p is recursively redundant if there is some k such that every tuple of t has at least one derivation tree that contains no more than k distinct p tuples.

Containment mapping imposes an order $\succeq$ on the strings in an expansion as follows: If there is a containment mapping from a string s_1 to a string s_2, then $s_1 \succeq s_2$. If there are containment mappings in both directions between s_1 and s_2, then we write $s_1 \equiv s_2$. A string s in the expansion S is a maximal ***string*** if there is no s' in S such that $s' \succeq$ s but not $s \succeq s'$. We define a ***containment sequence*** to be a sequence of strings, each string related to the next by a containment mapping.

A useful property that may hold in an expansion is containment-freedom: A set of strings S is ***containment-free*** if for any s in S, if s' maps to s, then s' = s.

Theorem 2.1 ***Let p be a predicate in a recursive rule r defining t, and suppose that in the expansion S oft, every containment sequence has a maximal element. Then p is redundant*** if ***and only if there is some k such that for any maximal string s in S, there is a string s', produced by eliminating all but k occurrences of p from s, such that s ≡ s'.***

Proof: Suppose that there is a k such that for each maximal string s_i in the expansion, we can remove all but k instances of p from s_i to produce s_i', and that $s_i \equiv s_i'$. Because there are containment mappings between s_i and s_i', s_i' defines the same relation as s;. Furthermore, because every containment sequence has a maximal element and by definition of "maximal," the relation for t is completely defined by the maximal strings. Then the relation for t can also be defined by the s_i', none of which contain more than k instances of p, and p must be redundant.

Conversely, if p is redundant, then t can be defined by some (possibly infinite) set of strings, none of which contains more than k instances of p. Let R be such a set of strings. Assume without loss of generality that R is containment-free.

Let s; be a maximal string in S. Construct a representative database ***edb*** as follows. Define a one-one mapping h from the variables in s_i to some set of constants. Then if $q(V_1, V_2, \ldots, V_n)$ appears in s_i, add the tuple $q(h(V_1), h(V_2), \ldots, h(V_n))$ to ***edb.*** By the definition of ***edb,*** if $D_1, D_2, \ldots, D_n$ are the distinguished variables in s_i, then h proves that the tuple $(h(D_1), h(D_2), \ldots, h(D_n))$ is in the relation returned by evaluating s_i over ***edb.***

Because R and S define the same relation, there must be some string r in R such that evaluating r over ***edb*** also returns $(h(D_1), \ldots, h(D_n))$. This implies that there is a mapping g from the variables in r to the constants in ***edb*** that maps tuples of variables appearing in a predicate p to tuples of constants appearing in the relation

for p. Because h is one-to-one, it is invertible. But then g o h^{-1} is a containment mapping from r to s;. (This is the result of Sagiv and Yannakakis [14] concerning equivalences between a pair of unions of tableaux: if the relation for a union U_1 contains the relation for a union U_2, then every string in U_2 must be mapped to by some string in U_1.) Let g' = g o h^{-1}.

By similar reasoning, there is some s_j in S such that there is a mapping m that maps Sj to r. But then m o g' maps s_j to s;. Because s_i is maximal, then either $s_j = s_i$ or there is mapping from s_i to s_j. Either way, there is a mapping m' from s_i to r.

Let s_i' denote the image of $g(r)$ in s_i. Because r has at most k instances of p, s_i' also has at most k instances of p. Since s_i' is a subsequence of s_i, the identity map maps s_i' to s_i. Also, by definition of s_i', m' o g' maps s_i to s_i'. Thus s_i' satisfies the conditions of the theorem. ∎

Corollary 2.1 ***Suppose that p appears in a recursive rule defining t and that the expansion S oft is containment-free. Then p is recursively redundant if and only if there is some k such that for every s in S, there is a string s', produced by eliminating all but k occurrences of p from*** s, ***such that*** $s_i \equiv s_i'$.

Proof: If an expansion is containment-free, then every string is a maximal string. The corollary follows from the theorem. ∎

Example 2.2 As a simple example, consider the rules r_2 and r_3 from the introduction. The expansion of the ***buys*** predicate, abbreviating each predicate by its first letter, begins

$$l(X,Y)c(Y),$$
$$k(X,W_0)l(W_0,Y)c(Y)c(Y),$$
$$k(X,W_0)k(W_0,W_1)l(W_1,Y)c(Y)c(Y)c(Y)$$

All but one of the instances of c(Y) can be removed from any string in the expansion, so c is recursively redundant. The expansion of rules r_4 and r_5 begins

$$l(X,Y)r(X),$$
$$k(X,W_0)l(W_0,Y)r(W_0)r(X),$$
$$k(X,W_1)k(W_1,W_0)l(W_0,Y)r(W_1)r(W_0)r(X)$$

This expansion is containment-free, and no predicate can be eliminated from any string, so r is not recursively redundant. ∎

Suppose that every predicate p in a recursive definition of a relation t is recursively redundant. Then, for each predicate p_i, there is a constant k_i such that no tuple of t depends on more than k_i tuples of p_i. This implies that t can be defined by a union of strings R such that for every i, no string in R contains more than k_i occurrences of p_i. Since there are only finitely many such strings, R must be a finite set.

But then R is a nonrecursive definition of t. This shows that the bounded recursion problem reduces to the redundant predicate problem. Recently, Gaifman has shown that the bounded recursion problem is undecidable [5], and various restrictions of the problem have been shown undecidable by Vardi [16] and by Mairson and Sagiv [9]. Thus we have proven the following:

Theorem 2.2 ***There is no algorithm that decides if a recursive definition is minimal.***

In spite of this somewhat discouraging result, we are able to give a linear-time algorithm that, while not identifying every recursively redundant predicate in a rule, never incorrectly identifies a predicate as recursively redundant. Furthermore, we can identify a useful class of rules for which the algorithm is complete.

3 The A/V Graph and Branches

In Section 2, we reduced the question of recursive redundance to minimality conditions on the strings in the expansion of the recursively defined predicate. These minimality conditions depend on certain properties of the strings. In this section we discuss those properties, and develop tools to detect them.

3.1 A/V Graphs and Expansions

This subsection summarizes part of a previous paper [11], so we omit the proofs here.

To relate the patterns of variables appearing in the strings of S to the structure of the rules, ***we*** define the ***argument/variable (A/V) graph:***

- For each variable appearing in the rules add a variable node.
- For each argument position in each rule body add an argument node.
- Draw an undirected edge from each argument node to the node for the variable that appears in that position in the rule. This kind of edge is called an ***identity edge.***
- Draw a directed edge from each argument node corresponding to a position p in the recursive predicate to the node for the distinguished variable that appears in the p in the rule head. This kind of edge is called ***a unification edge.***

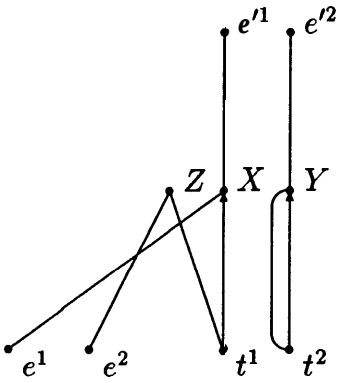

Figure 2: A/V graph for Example 2.1.

The node for a variable X is labeled X, and the node for argument position i of a predicate p is labeled $p:i$. A node for a distinguished variable is a distinguished variable node; all other variables nodes are nondistinguished. Because of the one-to-one correspondence between positions in the bodies of rules and the argument nodes in the A/V graph, we use position names to refer to both an argument position and the argument node it is represented by. Similarly, we use variable names to refer to variable nodes.

Many of the subsequent results depend on the existence of certain kinds of paths through the A/V graph. Some nonstandard terminology arises because we allow the directed edges in an A/V graph to be traversed from head to tail as well as from tail to head; thus a path in an A/V graph can contain unification edges traversed in either direction.

Example 3.1 Figure 2 gives the A/V graph for the rules of Example 2.1. ∎

There is a close relationship between the A/V graph and procedure ExpandRule of Section 2. If a predicate instance first appears through applying a rule on iteration i, then we say that predicate instance was produced on iteration i. (The first iteration of the while loop is iteration 0.) There are two ways a predicate appearing in a string s of S can be produced on iteration i. It can be added to CurString through applying the recursive rule, or, if s w-as added to S on iteration i, it can be produced by applying the nonrecursive rule.

. Consider iteration i. At line 8 on iteration $i-1$, the variables in the rules were given subscript i. Letting the argument nodes of the A/V graph represent the bodies of the rules, we represent iteration i by subscripting the labels of the variable nodes by i. (Figure 3(a)).

Because the heads of the rules contain no repeated variables or constants, the unification can be done by replacing the subscripted distinguished variables by the variables appearing in the instance of t in CurString. If we consider the argument nodes for t as representing that instance of t, the variable at the head of a unification edge is replaced by the variable appearing in the argument at the tail. On iteration 0, because of the initialization of

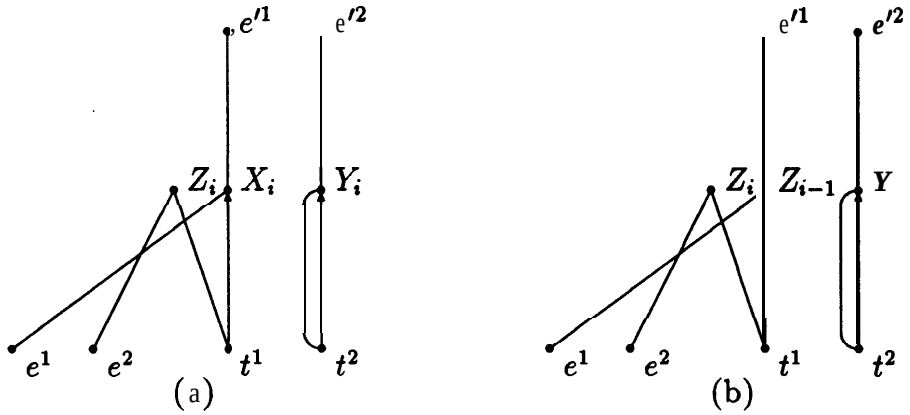

Figure 3: A/V graph for Example 2.1.

CurString, the arguments contain the distinguished variables. On all other iterations, they hold the variables that were put there on the previous iteration — in this case, Z_{i-1} and Y. (Figure 3(b)).

After the substitution, argument ***a*** of a predicate instance produced on iteration i will contain the variable that is the label of the node at the end of its incident identity edge. In our example, the predicate instance added by the nonrecursive rule will be $e'(Z_{i-1}, Y)$. and the predicate instances added by the recursive rule will be $e(Z_{i-1}, Z_i)t(Z_i, Y)$.

The previous two paragraphs show how we can determine what variable appears in any position of any predicate instance in the expansion. The following two facts can be proven by induction:

Fact 3.1 A nondistinguished variable W_i appears in position ***p*** in a predicate instance produced on iteration i + ***k*** if and only if there is a path from ***W*** to ***p*** containing ***k*** unification edges, all traversed in the forward direction.

Fact 3.2 A distinguished variable ***V*** appears in position ***p*** on iteration i if and only if there is a path from ***V*** to ***p*** containing i unification edges, all traversed in the forward direction.

Any A/V graph can be divided into two kinds of connected components, those containing *nondistinguished variables and those containing only distinguished variables. (Connected components in A/V graphs can require unification edges to be traversed in either direction.) Each type of component has a specific structure.

Lemma 3.1 ***If a connected component in an A/V graph contains a nondistinguished variable W, it is a tree, and W is the only nondistinguished variable in the component.***

Lemma 3.2 ***If a connected component contains no nondistinguished variable, that component must contain a cycle.***

Lemmas 3.1 and 3.2 combine with Facts 3.1 and ***3.2*** to prove that

1. Arguments in connected components that contain a cycle will eventually contain only the distinguished variables appearing on the cycle.

2. Arguments in connected components that contain no cycles will eventually contain only subscripted instances of the nondistinguished variable in the component.

Example 3.2 See Figure 2 for the A/V graph for Example 2.1. There are two connected components in this graph. The first, $\{t^2, Y, e'^2\}$, contains the cycle $t^2 \rightarrow$ Y $\rightarrow t^2$. Then Fact 3.2 implies that Y always appears in e'^2. The remaining nodes form a tree, with Z at the root. By Fact 3.1, Z_i appears in e^2 and e'^1 on iteration i, and in e^1 on iteration i + 1. I

In the following Subsection, it will be important to know how variables are shared between the predicate instances in the expansion. Things are complicated by the possibility of repeated variables in the rule body. Repeated variables give rise to branches in the paths from variable nodes to argument nodes, and the variable at the root of such paths appears on all branches. Thus to determine when argument positions share variables, we need to follow unification edges backward (toward the argument nodes) as well as forward.

To count the net number of forward unification edges in a path, we introduce weights on the edges of the A/V graph. The weight of an identity edge is 0; the weight of a unification edge traversed in the forward direction is 1, and the weight of a unification edge traversed in the reverse direction is -1. The weight of a path in the A/V graph is the sum of the weights of the edges in the path. With this definition, we have the following lemma:

Lemma ***3.3 FOT*** $i \geq max(j,$ ***k),*** ***a variable appears in position*** p_1 ***of a predicate instance produced on iteration*** $i + j$***, and in position*** p_2 ***of a predicate instance produced on iteration*** $i + k$***, if and only if there is a path from from*** p_1 ***to*** p_2 ***of weight*** $k - j$***.***

The period of the rule, τ, can be determined from the A/V graph for the rule.

Lemma ***3.4 If there are cycles in the A/V graph for the rule,*** τ ***is the least common multiple of the weights of the cycles; otherwise it is I.***

3.2 Chains and Branches

In this subsection we discuss some properties of the strings of an expansion that will be useful in determining redundancy. These properties depend only on the recursive rule.

Definition 3.1 A ***chain*** is a sequence of predicate instances $p_1, p_2, \ldots, p_n$, such that for $1 \leq$ i $<$ ***n,*** p_i and p_{i+1} share a nondistinguished variable.

Definition 3.2 A ***branch*** is a sequence of predicate instances such that for all ***p*** and ***q*** in the sequence, there is a chain containing ***p*** and ***q.***

The positions in which the shared nondistinguished variables appear are called ***linking positions.***

Example 3.3 Assuming that X and Y are the only distinguished variables,

$$p(X, W_0)p(W_0, W_1)p(W_1, Y)$$

is a chain of length three. The sequence

$$p(W_0, X)p(X, Y)p(Y, W_1)$$

contains three chains of length one. The sequence

$$p(X, w_0, Z_0)q(W_0)p(Z_0, W_1, Z_1)q(W_1)p(Z_1, W_2, Z_2)q(W_2)$$

is a branch. ∎

By examining the A/V graph for a recursive rule, we can tell what chains (and therefore, what branches) will appear in the strings of the expansion. Chains depend only on shared , nondistinguished variables, so the first task is to eliminate from the A/V graph the nodes for arguments that contain only distinguished variables. By Lemmas 3.1 and 3.2, we can do this by removing all connected components that contain cycles.

For the second step, we augment the remaining subgraph by adding predicate edges between adjacent argument nodes of the same nonrecursive predicate. The result is the ***augmented*** A/V graph for the rule.

Example 3.4 The following rule illustrates the concepts that will be developed in this section.

r_r: $t(X, Y) :- t(X, V), p(X, W), q(W, V), r(X, Y).$

The A/V graph for this rule is given in Figure *4.* X appears in a connected component with a cycle, so we remove the nodes for X, t^1, p^1, and r^2 The augmented A/V graph for the rule is given in Figure 5. ∎

The following lemma and its two corollaries show the relationship between the augmented A/V graph for a rule and the branches that appear in its expansion. Recall that in paths in an A/V graph, unification edges (arcs) can be traversed in either direction.

Lemma ***3.5 There is a chain containing an instance of predicate*** r ***produced on iteration i, and an instance of s produced on iteration*** $i + k$, ***if and only if there*** is ***a path of weight k from the arguments of r to the arguments of s.***

Proof: Given in [11]. ∎

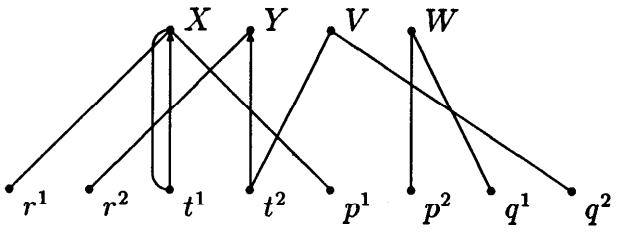

Figure 4: A/V graph for Example 3.4

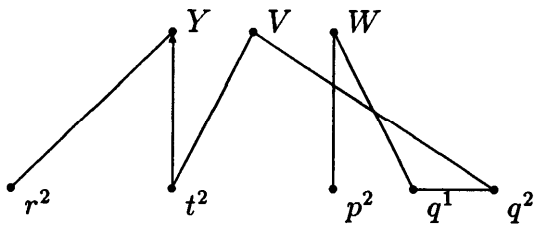

Figure 5: Augmented A/V graph for Example 3.4

Corollary ***3.1 Instances of two predicates appear in the same branch if and only*** if ***the argument nodes for those predicates appear in the same connected component.***

Proof: By definition, two predicates appear in the same branch if and only if there is a chain connecting them. ∎

Definition ***3.3*** The ***rank*** of a predicate is the weight of a maximal path to an argument node of the predicate. The ***span*** of a branch is the rank of its maximal predicate.

Corollary ***3.2 The span of a branch is the maximum weight of any path in the connected component for the branch.***

Proof: Immediate from the definition of span. ∎

Two branches in a string are ***instances of the same branch*** if the predicates in each appear in the same connected component in the augmented A/V graph. A rule produces multiple branches if there are multiple connected components in the augmented A/V graph for the rule. An instance of each branch is begun on every iteration. If the span of a branch is ***k,*** the first complete instance of that branch will be produced by iteration ***k*** + 1. In addition to the complete instances of branches, there will also be incomplete instances of branches produced.

Example 3.5 In Figure 5, there is only one connected component, and it contains all the argument nodes. This implies that there is a branch (chain, in this case) containing

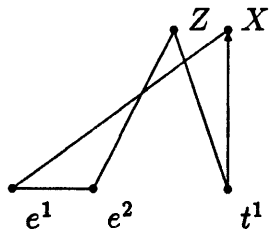

Figure 6: Augmented A/V graph for the recursive rule of Example 2.1.

instances of ***p, q,*** and ***r.*** There are paths of weight 0 from ***V*** and ***W*** to the nodes of ***p*** and ***q,*** thus ***p*** and ***q*** are of rank 0. There is a path from ***V*** to r^2 of weight 1, so ***r*** is of rank **1.** The span of the branch is 1.

The third string in the expansion,

$$p_2(X, W_1)q_2(W_1, V_1)r_2(X, V_0)p_1(X, W_0)q_1(W_0, V_0)r_1(X, Y),$$

contains a partial instance of the branch produced on iteration 1 $(r_1(X, Y))$, a complete instance produced on iterations 1 and 2 $(p_1(X, W_0)q_1(W_0, V_0)r_2(X, V_0))$, and another partial instance produced on iteration **2** $(p_2(X, W_1)q_2(W_1, V_1))$. This last partial instance will be completed on iteration 3. ▮

If there is a cycle of nonzero weight in a component of an augmented A/V, then there will be a branch with an infinite span. Such branches will never be completed; in this case we say the expansion contains ***unbounded branches.***

Definition ***3.4*** A connected component of an augmented A/V graph is a ***bounded component*** if it contains no cycle of nonzero weight; otherwise it is an ***unbounded component.***

Example 3.6 The augmented A/V graph for the recursive rule of the transitive closure example (Example 2.1), Figure 6, contains an unbounded component. The expansion contains unbounded branches — for any ***n,*** we can find a string in the expansion containing a branch of at least ***n*** instances of e. ▮

4 Testing for Redundancy

In this section we show that if a predicate ***p*** appears in a bounded component of the augmented A/V graph, then ***p*** is recursively redundant, and that for a useful subset of recursive rules, only predicates appearing in bounded components are redundant. These predicates can be found by a linear-time algorithm.

4.1 A Sufficient Condition

The following theorem relates redundancy to connected components in the augmented A/V graph for the recursive rule. (Recall that in a connected component of an A/V graph, unification edges can be traversed in either direction.)

Theorem 4.1 ***Let a predicate p appear in a rule r, and suppose that no argument of p appears in an unbounded component of the augmented A/V graph for r. Then p is recursively redundant in r.***

Proof: If no argument of p appears in an unbounded component of the augmented A/V graph, then either no argument of p appears in the augmented A/V graph, or every argument appears in a bounded component. These two cases are covered by the following two lemmas. ∎

Lemma ***4.1 Let a predicate p appear in a recursive rule r, and suppose that no argument node of p appears in the augmented A/V graph. Then p is recursively redundant in r.***

Proof: If no node of p appears in the augmented A/V graph, then all arguments of p must appear in cyclic components of the (unaugmented) A/V graph. Then all arguments of all instances p will contain only distinguished variables. These appear in the same positions every τ iterations; thus all but the first τ occurrences of such a predicate can be removed from any string of the expansion. ∎

Lemma 4.2 ***Let C be a bounded component of the augmented A/V graph for a recursive rule r. Then all predicates p appearing in C are recursively redundant in r.***

Proof: By Theorem 2.1, it will suffice to show that all but a constant number of the predicates in bounded components can be removed from any string of the expansion. Let τ be the period of the rule, and let σ be the maximum weight path in C. Suppose that p appears in C, and that p is of rank j. Claim: in string s_n, $n \geq \tau + \sigma$, we can remove all instances of p except

- Those produced on iteration 0 through iteration $\tau + j - 1$, and
- Those produced on iteration $n - (a - j)$ through $n - 1$.

Let s' be the result of doing this for every predicate appearing in C. If b_C is the branch corresponding to the component C, s' will be s with some complete instances of b_C removed. This will leave $\tau + \sigma$ instances of each predicate in C. We claim that there are containment mappings between s' and s.

The identity map maps s' to s. We can map s to s' by the containment mapping h defined as follows:

1. If V is a distinguished variable, $h(V) = V$,

2. If W_i is a nondistinguished variable not appearing in a removed predicate instance, $h(W_i) = W_i$ (a removed predicate instance is one that appears in s but not in s'.)

3. Let W_i be a nondistinguished variable appearing in a removed predicate instance of rank j that appeared in s on iteration $m\tau + k + j$, $m > 0$ and $0 \leq k < \tau$. Then if W_l is the variable that appeared in the corresponding position on iteration $k + j$, $h(W_i) = W_l$.

The predicate instances in s excluding those appearing in removed instances of b_C are mapped by h to themselves. If a predicate p is of rank j, then for all $m > 0$ and all k, $0 \leq k < \tau$, an instance of p produced on iteration $m\tau + k + j$ is isomorphic to the instance that was produced on iteration $k + j$. Thus the removed instances of b_C are mapped to isomorphic instances of the branches that were not removed.

Any of the first $\sigma + \tau$ strings produced will contain fewer than $\sigma + \tau$ instances of any predicate. By the above construction, for all p in C, in any later string we can remove all but $\sigma + \tau$ instances of p. Then p is recursively redundant in r. ∎

Example 4.1 Consider the following rules:

$$t(w, x, Y, Z) :- t_0(W, x, Y, Z).$$
$$t(W, X, Y, Z) :- t(X, W, P, Q), e(P, Y), a(X, Q), b(Z).$$

In the A/V graph for the recursive rule (Figure 7), there is a cycle involving X and W of weight 2, thus $\tau = 2$.

In the augmented A/V graph for the recursive rule there are two components. The argument nodes of e appear in an unbounded component, while all other predicate arguments appear in a component with a maximal path of weight 1. The rank of a is 0, and the rank of b is 1. Consider the string produced on iteration five. We display it below, one branch to a line, one column for each iteration, starting with iteration 0 on the right.

1)	$t_0(X, W, P_4, Q_4)$	$a(X, Q_4)$	$e(P_4, P_3)$	$e(P_3, P_2)$	$e(P_2, P_1)$	$e(P_1, P_0)$	$e(P_0, Y)$
2)		$b(Q3)$	$a(W, Q3)$				
3)			$b(Q_2)$	$a(X, Q_2)$			
4)				$b(Q_1)$	$a(W, Q_1)$		
5)					$b(Q_0)$	$a(X, Q_0)$	
6)						$b(Z)$	

Following the construction of the proof of Lemma 4.2, we generate s'_5 by deleting the a and b predicates appearing on lines 2 and 3. The string s_5 can be mapped to s'_5 by mapping the distinguished variables W, X, Y, and Z to themselves, Q_0, Q_1, and Q_4 to themselves, Q_3 to Q_1, and Q_2 to Q_0. This leaves 3 instances of a and b. ∎

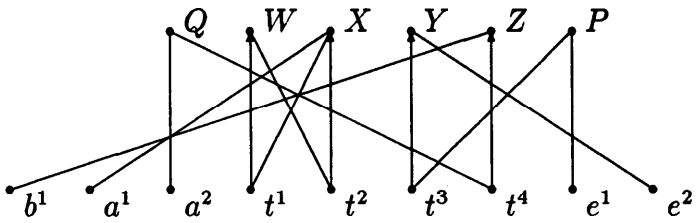

Figure 7: A/V graph for the recursive rule of Example 4.1

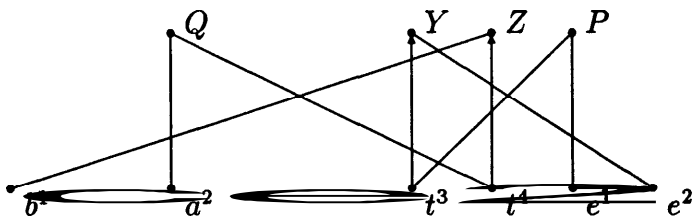

Figure 8: Augmented A/V graph for the recursive rule of Example 4.1

4.2 A Necessary and Sufficient Condition

In general, the converse of Theorem 4.1 fails to hold: some predicates appearing in unbounded components are redundant. If the nonrecursive predicates are IDB predicates, their definition may cause redundancy. For example, if we take the standard transitive closure rules,

t(X,Y) :- e(X,Y).

t(X, Y) :- t(X, W), e(W, Y).

and add

e(X, Y) :- a(X), b(Y).

then *t* is completely defined by the rule

t(X, Y) :- a(X), b(Y).

and e is redundant in the recursive rule.

Even if the nonrecursive predicates are EDB predicates, interactions between the recursive and nonrecursive rule can make predicates redundant. The pair of rules

t(X, Y) :- b(X), t(X, W), e(W, Y).

t(X, Y) :- b(X), e(W, Y).

can be replaced by the nonrecursive rule alone.

Recently, Vardi [16] has shown that dealing explicitly with the nonrecursive initialization makes the bounded recursion problem undecidable, even for the special class of recursive definitions containing only one linear recursive rule. Because the bounded re-

cursion problem reduces to the minimal recursive definition problem, detecting recursively redundant predicates is also undecidable for this class of definitions. However, if we require initialization by a single, arbitrary rule, the following theorem holds.

Theorem 4.2 ***Let all*** nonrecursive ***predicates*** in ***r be EDB predicates, and suppose that there are no repeated nonrecursive*** predicates in ***r. Then a predicate p*** is ***recursively redundant in r*** if ***and only if p appears in a bounded component of the augmented A/V graph.***

Proof: The "if" direction is given by Theorem 4.1.

For the other direction, suppose that b is a branch corresponding to a connected component C, and that C contains a nonzero weight cycle. Because C is unbounded, b will be an unbounded branch. Define a level zero predicate instance in b be one such that there is no predicate instance in b appearing on an earlier iteration. For $k > 0$, p is a level k predicate instance in b if p shares a nondistinguished variable with a level $k-1$ predicate instance in b.

To complete the proof we use two facts:

1. In any unbounded branch, there is a level zero predicate instance that contains a distinguished variable in a linking position.

2. No variable appears twice in any linking position of any predicate in an unbounded branch.

(The proofs of these facts are straightforward modifications of the proofs of Facts 4.1 and 4.2 in an earlier paper [11]).

Suppose that h maps s to some subsequence s' of itself. We show by induction on level that for all i, if a variable W appears in a level i predicate of instance of b, then $h(W) = W$.

For the basis, i = 0, by the first fact there is a level zero predicate instance p that contains a distinguished variable V in a linking position. Because there are no repeated predicates in the body of the rule, the second fact implies that this is the only instance of p in s that contains V in that position. Then by definition of a containment mapping, for all variables W in that instance of p, $h(W) = W$.

Consider some level k predicate instance p. By definition of level, p shares some nondistinguished variable W_i with a level k-l predicate. By induction, $h(W_i) = W_i$. Suppose that W_i appears in position p^1 in p. Again by the second fact, this instance of p is the only one in which W_i appears in p^1. But then for all variables W_j in p, $h(W_j) = W_j$.

Suppose s' is generated by removing some predicate instance p from s. Let W be a nondistinguished variable appearing in a linking position p^1 of that instance of p. By the above, if h is to map s to s', then $h(W) = W$. But because there are

no repeated predicates in the rule body, and by the second fact, there is no other instance of p that contains W in position p? So s cannot map to s'.

By another result from [11] we know that in every string of the expansion, some position t_0^1 in the predicate from the nonrecursive rule will share some nondistinguished variable W with a predicate instance in b. This variable W will be different in every string. By an argument similar to the above, we can prove that if h is to map a string s_1 to another string s_2 in the expansion, $h(W) = W$. But then h cannot map the instance of t_0 in s_1 to the instance in s_2.

This shows that the expansion is containment-free, and that no instance of p can be removed from any string, so by Corollary 2.1, if p appears in an unbounded component, p is not redundant. ∎

4.3 Detecting Redundant Predicates

In this subsection we present a linear-time algorithm that detects redundant predicates in an augmented A/V graph. A variant of this algorithm was presented without proof in Ioannides [7]. There it was used on a different graph, the a-graph, to decide the bounded recursion problem for a restricted class of rules.

An A/V graph can be converted to an acyclic A/V graph by a straightforward application of depth-first search. We assume that the augmented A/V graph is presented in adjacency list form. Each edge is represented by a pair of directed edges. Associated with each directed edge is a weight. Both directed edges for a predicate or identity edge have weight zero; the directed edge corresponding to a forward traversed unification edge has weight one, while the directed edge for a reverse unification edge has weight minus one.

To prove the correctness of the algorithm, we consider two cases, depending on whether procedure SearchComp (Figure 9) is called on a vertex of a bounded or unbounded component .

Lemma *4.3 If* ***SearchComp is called on a bounded component, SearchComp returns with cycle set to false.***

Proof: Suppose that SearchComp is called on a node of a bounded component, and that in the course of searching that component, it visits a node n twice, with a different weights each time. Then there must be two paths from the start node to n. These paths start from the same node, so there must be some shared initial segment. Let n' be the last node of this shared initial segment, and let the two paths from n' to n be denoted p_1 and p_2. Because the weight of n differs along these two paths, the weights of p_1 and p_2 must differ. Because every path in an A/V graph can be traversed in opposite directions at opposite costs, the concatenation of p_1 with p_2 reversed would be a cycle of nonzero weight, which is a contradiction. Thus every time a node is visited, it must receive the same weight, and line *4* of

Input. An augmented A/V graph G = ***(V,*** E) represented by adjacency lists $L[v]$, for v ∈ ***V.*** The function ***w(e(v,u))*** returns the weight for each edge e(v, u).
output. A list of the bounded components in the graph.

```
cycle:    boolean;          /* true if current component contains a cycle */
w:        array of integer; /* w[u] holds weight of u if visited */

Procedure SearchComp( v, cycle);

begin
1.    mark v old;
2.    for each vertex u on L[v] do
3.        if u is marked "old" and w[u] ≠ weight + w( e(v, u)) then
4.            cycle := true;
5.            return;
6.        else
7.            weight := weight + w(e(v, u));
8.            w[u] := weight;
9.            SearchComp( v);
10.       endif;
11.   endfor;
end;

begin
1.    mark all vertices "new";
2.    while there exists a vertex v in V marked "new" do
3.        cycle := false;
4.        SearchComp( v);
4.        if not cycle then
6.            PrintComponent(
7.        endif;
8.    endwhile;
end.
```

Figure 9: An algorithm to detect redundant predicates.

SearchComp is never executed, and the initial call to SearchComp will return with cycle set to false. ▮

Lemma 4.4 *If SearchComp is called on an unbounded component, SearchComp returns with cycle set to true.*

Proof: Suppose that there is a cycle of nonzero weight, but that SearchComp doesn't find it. SearchComp must reach some node of the cycle first. Let n_1 be that node, and number the subsequent nodes in the cycle $n_2, \ldots, n_m$, where there are m nodes in the cycle. Let the weight of the path from the root of the depth-first spanning tree to n_1 be w_1. If SearchComp continues around the cycle in order, it will again reach n_1, this time with a weight $w_1 + \sum_{i=1}^{m-1} w(e(n_i, n_{i+1})) \neq w_1$, and SearchComp will return with cycle set to true. Let the weights assigned to n_1 through n_m by following the cycle in order be w_1 through w_m. If SearchComp is to miss the cycle, it must search the component in some other order.

In this new order, the weight assigned to at least one node in the cycle must differ from the weight would be assigned if SearchComp searched around the cycle in order. Let n_i be the first such node in the cycle, that is, the depth-first search assigns n_{i-1} weight w_{i-1}, but node n_i weight $w_i' \neq w_i$. Then the node preceding n_i in the depth-first search must not be n_{i-1}, or else n_i would have received weight $w_{i-1} + w(e(n_{i-1}, n_i)) = w_i$. But then when the edge from n_i to n_{i-1} is traversed, the current weight $w_i' + w(e(n_i, n_{i-1})) = w_i' + -w(e(n_{i-1}, n_i))$ will not equal w_{i-1}, and a nonzero weight cycle will again be detected. ▮

Lemmas 4.3 and 4.4 combine to show that the algorithm outputs exactly the predicates that appear in bounded components. By Theorem 4.1, these predicates are redundant.

5 Optimizing Recursive Definitions

In this section we present an algorithm for optimizing recursive definitions. We assume that the algorithm has available:

1. σ, the maximal span of any bounded component in the augmented A/V graph for the rule.
2. For each connected component C of the augmented A/V graph, the span σ_C of that component and a list of predicates appearing in that component.
3. A list of predicates that apear in the rule but not in the augmented A/V graph.
4. τ, the period of the recursive rule.

The first two items can be computed by a straightforward modification of the algorithm of the previous section. A depth-first search will find the components of the (unaugmented) A/V graph that contain cycles, and the weights of these cycles. (This uses the fact that each component of an A/V graph contains at most one cycle.) This can be used to compute τ, and to produce the list of predicates in item 3.

The optimization converts a set of rules R to a new set of rules R'in which recursively redundant predicates have been removed from the recursive rule. It works as follows: Generate the first $\sigma + \tau$ strings of the expansion of R. Add to R' the rules

r_i: $_{\iota} :- s_i$.

for $0 \leq i < \sigma + \tau$.

The remaining rules in R' are constructed from string $s_{\sigma+\tau}$. Let the set P_1 contain the instances of predicates p in $s_{\sigma+\tau}$ such that

- p appears in a connected component C of finite span σ_C,
- pisofrank j,
- p was produced on iteration i, $\square \leq i \leq \tau$ + j $-$ 1.

Let the set P_2 contain the instances of predicates p in $s_{\sigma+\tau}$ such that

- p doesn't appear in the augmented A/V graph for r, and
- p was produced on the first τ iterations.

and let the set P_3 contain all the instances of predicates p in $s_{\sigma+\tau}$ such that

- p appears in a connected component C of finite span σ_C,
- pisofrank j,
- p was produced on iteration $i, \sigma + \tau - (\sigma_C -$ j) $\leq i \leq \sigma + \tau -$ 1.

Let the set Q be the instances of predicates in $s_{\sigma+\tau}$ that appear in unbounded components of the augmented A/V graph.

Create a new predicate $\blacklozenge$ where $\blacklozenge$ has the same number of arguments as $\blacklozenge$ and add the rule

$r_{\sigma+\tau}$: $\blacklozenge :- t', P_1, P_2, Q$.

where the instance of $\blacklozenge$ has the same variables as the instance of $\blacklozenge$ in string $s_{\sigma+\tau}$ of the expansion. Also add the rule

$r_{t_0'}$: $t' :- t_0, P_3$.

where t_0 contains distinguished variables as in the original nonrecursive rule, and where the nondistinguished variables in the instances in P_3 are replaced by distinguished variables as necessary so that the instances share variables with the same arguments of t_0 as they did in $s_{\sigma+\tau}$. Finally, add a new recursive rule

$r_{t'}$: $t' :- t', q_1, \ldots, q_n$.

where the body of $r_{t'}$ is the body of the original recursive rule, with t replaced by t', and with the predicates appearing in bounded components of the augmented A/V graph removed. For any variable V appearing in a position t'^i the head of $r_{t'}$ but not in the body, replace the variable appearing in t'^i in the body of $r_{t'}$ by V.

Theorem 5.1 *The new rules define the same relation as the old.*

Proof: Let s_i, for i $\geq$ 0, be the strings of the expansion of the original rules, and let s_i', for $i \geq 0$, be the strings of the expansion of the new rules. We prove the theorem by showing that for i $\geq$ 0, $s_i' \equiv s_i$.

For i < $\sigma + \tau$, $s_i' = s_i$, so, trivially, $s_i' \equiv s_i$. Consider some string s_i', where $i \geq \sigma + \tau$. The string s_i' was generated by one application of $r_{\sigma+\tau}$, i $-$ (a + τ) applications of $r_{t'}$, followed by one application of $r_{t_0'}$. We claim that there are containment mappings between s_i' and s_i.

For any predicate p appearing in unbounded branches, the first $\sigma + \tau$ instances of p appear in the body of rule $r_{\sigma+\tau}$, and the remaining instances will be generated by the i $-$ (a + τ) applications of rule $r_{t'}$. This means that the predicate instances in unbounded branches will be identical in s_i' and s_i.

Let p be a predicate of rank j appearing in a component C with a bounded span σ_C. By the construction in the proof of Lemma 4.2, all instances of p in s_i are redundant except for those produced on iterations 0 through τ + j $-$ 1, and on iterations i $-$ (σ_C $-$ j) and i $-$ 1. The first group appear in the set P_1, so they will also appear in rule $r_{\sigma+\tau}$, and therefore in string s_i'. Instances of predicates in the second group appear in the set P_3, and therefore in $r_{t_0'}$. After the unifications required to apply $r_{t_0'}$, these instances added to s_i' will be isomorphic to those in s_i.

The only other predicates in s_i are those that do not appear in the augmented A/V graph. By the construction in the proof of Lemma 4.1, all instances of these predicates can be removed except those appearing on the first τ iterations. But these predicate instances are group P_2, and therefore appear in $r_{\sigma+\tau}$ and also in s_i'.

Thus the strings s_i' are isomorphic to the strings produced by removing redundant predicates from the s_i as in the proofs of Lemmas 4.1 and 4.2, so for all $i \geq 0$, $s_i' \equiv s_i$. *I*

Example 5.1 We return to the rules of Example 4.1,

$$t(W,X,Y,Z) :- t_0(W,X,Y,Z).$$
$$t(W,X,Y,Z) :- t(X,W,P,Q), e(P,Y), a(X,Q), b(Z).$$

Here τ = 2, and there is only one bounded branch, so $\sigma = \sigma_C$ = 1. String $s_{\sigma+\tau}$ is string s_3, the fourth string generated. The first four strings of the expansion are

$$t_0(W,X,Y,Z)$$
$$t_0(X,W,P_0,Q_0)e(P_0,Y)a(X,Q_0)b(Z)$$
$$t_0(W,X,P_1,Q_1)e(P_1,P_0)a(W,Q_1)b(Q_0)e(P_0,Y)a(X,Q_0)b(Z)$$
$$t_0(X,W,P_2,Q_2)e(P_2,\ P\&(X,\ Q_2)b(Q_1)e(P_1,\ P_0)a(W,Q_1)b(Q_0)e(P_0,Y)a(X,\ Q_0)b(Z)$$

Then $P_1 = \{b(Q_1), a(W,Q_1), b(Q_0), a(X,Q_0), b(Z)\}$, P_2 is empty, $P_3 = \{a(X,Q_2)\}$, and $Q = \{e(P_2,P_1), e(P_1,P_0), e(P_0,Y)\}$. The new rules are:

r_0: $t(W, X, Y, Z) :- t_0(W, X, Y, Z).$
r_1: $t(W, X, Y, Z) :- t_0(X, W, P_0, Q_0), e(P_0, Y), a(X, Q_0), b(Z).$
r_2: $t(W,X,Y,Z) :- t_0(X,W,P_1,Q_1), e(P_1,P_0), a(W,Q_1), b(Q_0), e(P_0,Y), a(X,Q_0), b(Z).$
r_3: $t(W, X, Y, Z) :- t'(X, W, P_2, Q_2), e(P_2, P_1), e(P_1, P_0), e(P_0, Y),$
$b(Q_1), a(W,Q_1), b(Q_0), a(X,Q_0), b(Z).$

$r_{t'_0}$: $t'(W, X, Y, Z) :- t_0(W,X,Y,Z), a(W,Z).$
$r_{t'}$: $t'(W,X,Y,Z) : - \quad t'(X,W,R,Z), e(R,Y).$

(Here for clarity we have renamed the nondistinguished variable appearing in rule $r_{t'}$.) The sixth string generated by the new rules, displayed one branch per line and with predicates grouped by the iteration on which they appeared, is

1)	$t_0(X,\ W,R_1,Q_2)$	$a(W,Q_2)e(R_1,R_0)$	$e(R_0,P_2)$	$e(P_2,P_1)$	$e(P_1,P_0)$	$e(P_1,Y)$
2)				$b(Q_1)$	$a(X,Q_1)$	
3)					$b(Q_0)$	$a(X,Q_0)$
4)						$b(Z)$

This string is isomorphic to the sixth string of the original rules, after removing the redundant complete branches. I

Theorem 5.2 *If the original recursive rule contained no repeated nonrecursive predicates, then -the new recursive rule $r_{t'}$ is minimal.*

Proof: If the original recursive rule contained no repeated nonrecursive predicates, then the algorithm presented in Subsection 4.3 will be a correct and complete procedure for detecting redundant predicates. The transformation above removes all predicates that algorithm says are redundant; thus $r_{t'}$ must contain no recursively redundant predicates. ▌

Although the new recursive rule is minimal,, the new nonrecursive rules may not be minimal as tableaux. There are two reasons for this. First, if the rule produces multiple

bounded branches, not all branches will have an upper bound of σ. Second, if the original nonrecursive rule already contains an instance of a recursively redundant predicate, it may interact with the recursive rules to create redundancy.

Because only the nonrecursive rules can be non-minimal, the new set of rules generates an expansion in which each string contains at most a small constant number of redundant predicates. This is a great improvement over the original rules, which can generate expansions with arbitrarily many redundant predicates. The bodies of the nonrecursive rules are relational expressions, so to further eliminate redundancy they can be minimized by the tableaux techniques developed by Aho et. al [1].

Example 5.2 Here are the rules introduced in the introduction, abbreviating each predicate by its first letter.

$b(X, Y) :- l(X, Y), c(Y).$
$b(X, Y) :- k(X, Z), b(Z, Y), c(Y).$

For these rules, $\tau = 1$, and $\sigma = 0$. P_1 and P_3 are empty, $P_2 = \{c(Y)\}$, and Q $= \{k(X, Z_0)\}$. The new rules are

r_0: $b(X, Y) :- l(X, Y), c(Y).$
r_1: $b(X, Y) :- k(X, Z_0), b'(Z_0, Y), c(Y), c(Y).$

$r_{b'_0}$: $b'(X, Y) :- l(X, Y).$
$r_{b'}$: $b'(X, Y) :- k(X, Z), b'(Z, Y).$

Rule r_1 contains a redundant c predicate, so each string of the expansion will also contain a redundant c predicate. They can be removed by optimizing the body of the r_1 as a tableau. ∎

The number of rules added by this transformation will be $\sigma + \tau$. The constant σ is bounded above by the number of arguments in the recursive predicate. The constant τ is equal to the least common multiple of the weights of cycles in the A/V graph. It is difficult to give an analytic expression for τ, but in the worst case it can grow exponentially in the number of arguments in the recursive predicate. However, even if there are 6 arguments, $\tau \leq 6$. In practice, people rarely write predicates of many arguments. In fact, a reasonable assumption is that the arity of the predicates in the system is bounded, so we expect that the number of rules added will be small in any realistic application.

Most of the proposed evaluation algorithms for recursive rules can be viewed as techniques for evaluating the expansion of the rules. If the algorithm does not cache results of subgoals and check each new subgoal to see if it has already been solved, it will repeatedly solve redundant subgoals. If the same algorithm is run on the optimized rules, it will avoid this redundant computation, without any run-time overhead.

Acknowledgement: I'd like to thank John Hershberger, Yehoshua Sagiv, and Jeffrey Ullman for many helpful discussions on this material.

References

[1] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalence of relational expressions. ***SIAM Journal of Computing,*** 8(2):218–246, 1979.

[2] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In ***Proceedings of the Sixth ACM Symposium on Principles of Programming Languages,*** pages 110-120, 1979.

[3] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In ***Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,*** 1986.

[4] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In ***Conference Record of the Ninth Annual ACM Symposium on Theory of Computing,*** pages 77-90, 1977.

[5] Haym Gaifman. January 1986. NAIL! seminar, Stanford University.

[6] Lawrence J. Henschen and Shamin A. Naqvi. On compiling queries in recursive first order databases. *JACM*, 31(1):47–85, 1984.

[7] Yannis E. Ioannides. ***Bounded recursion in deductive databases.*** Technical Report UCB/ERL M85/6, UC Berkeley, February 1985.

[8] Michael Kifer and Eliezer L. Lozinskii. A framework for an efficient implementation of deductive databases. 1985. Unpublished manuscript.

[9] Harry G. Mairson and Yehoshua Sagiv. February 1986. NAIL! seminar, Stanford University.

[10] Jack Minker and Jean M. Nicolas. On recursive axioms in relational databases. ***Information Systems,*** 8(1):1–13, 1982.

[11] Jeffrey F. Naughton. Data independent recursion in deductive databases. In ***Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,*** 1986.

[12] Raymond Reiter. On closed world databases. In Herve Gallaire and Jack Minker, editors, ***Logic and Databases,*** pages 55-76, Plenum Press, New York, 1978.

[13] Yehoshua Sagiv. On computing restricted projections of representative instances. In ***Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,*** pages 171-180, 1985.

[14] Ye ɒS ha S ●g1 v and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *JACM*, 27(4):633–655, October 1980.

[15] Alle nV aG eIder. *A Message Passing Framework for Logical Query Evaluation.* Technical Report STAN-CS-85-1088, Stanford University, 1985.

[16] Moshe Y. Vardi. February 1986. NAIL! seminar, Stanford University.