February 1986

Report No. STAN-CS-86-1102

Data Independent Recursion in Deductive Databases

by

Jeff Naughton

Department of Computer Science

Stanford University
Stanford, CA 94305

Data Independent Recursion in Deductive Databases

Jeff Naughton*
Stanford University

February 28, 1986

Abstract

Some recursive definitions in deductive database systems can be replaced by equivalent nonrecursive definitions. In this paper we give a linear-time algorithm that detects many such definitions, and specify a useful subset of recursive definitions for which the algorithm is complete. It is unlikely that our algorithm can be extended significantly, as recent results by Gaifman [5] and Vardi [19] show that the general problem is undecidable. We consider two types of initialization of the recursively defined relation: arbitrary initialization, and initialization by a given nonrecursive rule. This extends earlier work by Minker and Nicolas [10], and by Ioannidis [7], and is related to bounded tableau results by Sagiv [14]. Even if there is no equivalent equivalent nonrecursive definition, a modification of our algorithm can be used to optimize a recursive definition and improve the efficiency of the compiled evaluation algorithms proposed in Henschen and Naqvi [6] and in Bancilhon et al. [3].

1 Introduction

In order to increase the expressive power of database systems, many authors have proposed augmenting these systems with logic-based query languages. These languages can be viewed either as an extension of relational query languages or as restricted logic programming languages. The result is sometimes called a deductive database, because of the ease with which implicit information can be "deduced" from the facts stored in the relations.

Our model follows that of Reiter [13], and consists of two parts. 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

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

the EDB. These inference rules are function-free Horn clauses that contain no negation or equality. An IDB predicate is recursive if its definition depends, directly or indirectly, on itself.

Recursive rules make logic-based query languages strictly more powerful than relational languages. This extra power is not free — recursion can be a major source of inefficiency. There is currently a great deal of interest in finding efficient evaluation algorithms for recursive rules (see for example [3,6,8,16,18].) Here we investigate properties of recursive definitions that are independent of the particular evaluation algorithm used.

In the absence of recursion, Reiter [12] has shown that the definition of an IDB relation can be "compiled" to a disjunction of conjunctions of EDB relations. We will call this disjunction the ***expansion*** of the IDB predicate; it contains every conjunction of EDB predicates that can be generated by some sequence of rule applications to the IDB predicate.

The expansion of a recursively defined predicate is infinite. However, because there are no function symbols in the inference rules, there is no way to introduce new values into the system, and even recursively defined relations must be finite. Then for any given state of the EDB, only a finite subset of the expansion needs to be evaluated. In general, this subset depends on the data in the EDB relations, and the recursion is ***data dependent.*** But for some recursive rules, we can prove that evaluating a fixed subset of the expansion will suffice for arbitrary values of the EDB, and the recursion is ***data independent.***

There are two natural definitions of data independence. The first is that stated in the previous paragraph — a set of rules ***is data independent*** if and only if it can be replaced by a fixed, finite set of nonrecursive rules. The second definition is more restrictive, and examines only the recursive rules. A set of recursive rules is ***strongly data independent*** if and only if adding any nonrecursive rule produces a set that is data independent. Because any fixed set of nonrecursive rules is equivalent to some first order expression, a set of rules is data independent if and only if it is equivalent to some first order expression.

Example 1.1 If e is the edge relation for a digraph, the following rules define the transitive closure of the graph:

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

Aho and Ullman [2] prove that the transitive closure is not equivalent to any first order expression, so this pair of rules is not data independent. This in turn implies that the recursive rule is not strongly data independent. ∎

Example 1.2 Suppose we have an EDB relation *likes(X,Y)*, where *likes(X,Y)* means that person X likes product Y. Suppose we also have an EDB relation ***trendy(X)***, where ***trendy(X)*** means that person X is trendy. If we know that a person will buy a product if

they like it, or if they're trendy and someone else has bought it, we can find all consumers and the products they buy using the following rules.

buys(X, Y) :- likes(X, Y).
buys(X, Y) :- trendy(X), buys(Z, Y).

This pair of rules can be replaced by

buys(X, Y) :- likes(X, Y).
buys(X, Y) :- trendy(X), likes(Z, Y).

so the original pair of rules was data independent. ∎

Previous work [7,10,14] considers only strong data independence, although the two are not equivalent — it is possible for a data independent set of rules to include recursive rules that are not strongly data independent.

To further discuss related work we need a few definitions. Initially we consider a relation t defined by a linear recursive rule $t :- t, p_1, p_2, \ldots, p_n$, and a nonrecursive or exit rule $t :- e_1, e_2, \ldots, e_m$, where p and e are EDB predicates. (A *linear recursive rule* is a rule with exactly one recursive predicate.) We add the restriction that the rule heads contain no repeated variables and no constants.

Variables appearing in the heads of the rules ***axe distinguished variables,*** and those appearing only in rule bodies ***axe nondistinguished variables.*** We standardize the variables in the rules so that the rule heads are identical, and the nondistinguished variables in different rule bodies are disjoint.

Sagiv [14] presents a tableau-theoretic approach to the problem. He considers typed rules of a single predicate. (A rule is *typed* if each variable appears in exactly one argument of the predicate, although it may appear in several occurrences of that predicate.) His results can be interpreted as giving a necessary and sufficient condition for strong data independence for sets of rules in this class.

Minker and Nicolas [10] adopt a theorem-proving approach. They determine a class of recursive rules such that for any rule in the class, all branches of a resolution refutation for the predicate at the head of the rule can be terminated by subsumption. In our terminology, they give a sufficient condition for strong data independence. Their class of rules includes nonlinear recursion, but excludes all permutations of distinguished variables except in predicates in which no nondistinguished variable appears. In addition, they disallow shared nondistinguished variables between predicates.

The work by Ioannides [7] is the most similar to this work. He gives a necessary and sufficient condition for strong data independence in single linear recursive rules having the property that no subset of argument positions of the recursive predicate in the rule body contains a permutation of the variables appearing in the same positions in the rule head. This includes the trivial permutation, so the test doesn't apply to any rule in which a distinguished variable appears in the same position in the rule head and in the rule body.

We now describe the organization and principle results of the paper.

In Section 2, we give a procedure that enumerates the expansion of a recursively defined predicate. This section also establishes an equivalence between data independence and the existence of mappings between the strings of the expansion. Section 3 introduces the ***argument/variable graph,*** a graph that concisely represents information about the structure of these strings.

Section 4 defines ***chain generating paths*** in the argument/variable graph. We show that the absence such a path is a sufficient condition for a linear recursive rule to be strongly data independent, and show that it is a necessary and sufficient condition if the rule has no repeated nonrecursive predicates.

Also in Section 4, we show that simple data independence is not equivalent to strong data independence, and give a necessary and sufficient condition for a regular recursive-nonrecursive rule pair to be data independent. (The body of a regular recursive rule contains only one nonrecursive predicate.) Section 5 extends Section 4 by giving a sufficient condition for strong data independence in a set of linear recursive rules.

Finally, in Section 6, we show two ways that the techniques of Section 4 can be used to optimize the evaluation of recursive queries. The first simply notes that data independence implies that complex termination conditions can be replaced by iteration bounds; the second shows that, in the data dependent case, we can detect predicates that can be moved out of the recursion, just as loop-invariants can be moved out of loops in procedural programming languages.

2 Data Independence and Expansions

We begin this section by repeating the definitions of data independence.

Definition 2.1 A set of rules is ***data independent*** if it can be replaced by a fixed, finite set of nonrecursive rules.

Definition 2.2 A set of recursive rules is ***strongly data independent*** if adding any nonrecursive rule produces a data independent set.

We will call data independence of the first type "weak" data independence when it is necessary to distinguish between the two.

The expansion of an IDB predicate is the set of all conjunctions of EDB predicates that can be generated by some sequence of rule applications to that predicate. Data independence can be decided by investigating containments between the relations specified by the elements of an expansion.

The strings of the expansion of a recursive predicate t can be enumerated by systematically applying sequences of rules to t. Although the results of this section hold for arbitrary recursive definitions, for concreteness we show how to generate the expansion of a predicate defined by one linear recursive rule and one nonrecursive rule.

```
1)   Give all variables in rules subscript 0;
2)   S := 0;
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_e;
9)   endwhile;
```

Figure 1: Procedure ExpandRule.

Procedure **ExpandRule** (Figure 1) enumerates the expansion for predicates defined by linear recursive rules with heads that contain no repeated variables or constants. Since this procedure imposes an order on the predicates in the conjunctions, we refer to them as strings. The input **to ExpandRule is a recursive rule, r_r, and an exit rule, r_e.** The **output** is the expansion of the recursively defined predicate, represented by the infinite set S.

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 Here we repeat the rules from Example 1.1.

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

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

/

The strings of an expansion are conjunctive queries, which Chandra and Merlin [4] have shown to be a subset of relational expressions. 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

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

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

To decide containments between the relations for these strings we use techniques related to tableaux, a tool developed by Aho et al. [1] for deciding equivalences between relational expressions.

Definition 2.3 A mapping m from the variables of a string s_1 into the variables of a string s_2 is a ***containment mapping*** if m maps distinguished variables 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: Suppose that s_1 maps to s_2, and that tuple t is in the relation specified by s_2. Since t is in the relation specified by s_2, there is some mapping m_1 that maps tuples of variables in s_2 into tuples of data values in the database. In particular, m_1 maps the tuple $(V_1, \ldots, V_n)$ appearing in $p(V_1, \ldots, V_n)$ to the tuple $(a_1, \ldots, a_n)$ in the relation for p, and maps the distinguished variables of s_2 to the elements of t.

If m_2 is the mapping from s_1 to s_2, then, by definition of string mappings, $m_1(m_2(s_1))$ is a mapping from s_1 to the database that proves that t is in the relation specified by s_1. ∎

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

Definition 2.4 Two strings are *isomorphic* if they are identical up to renaming of nondistinguished variables.

There are containment mappings in both directions between isomorphic strings. For a given set of rules, the number of predicates, argument positions in predicates, and distinguished variables are all bounded. Thus for any k, there are only finitely many nonisomorphic strings of k predicates. To rephrase 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.

We can use Lemma 2.1 to relate data dependent recursion to mappings between the strings in expansion of the recursively defined predicate.

Theorem 2.1 *A set of rules defining a predicate t is data independent if and only if, in the expansion of t, there exists an n_0 such that for all $n > n_0$, s_n is mapped to by some previous string.*

Proof: Suppose there is such an n_0. By Lemma 2.1, after evaluating the first n_0 strings, evaluating subsequent strings can return no new tuples, and the first n_0 strings completely define the relation. If s_i, $0 \leq i \leq n_0$, are the first n_0 strings of the expansion, then the recursive definition can be replaced by the n_0 rules $t :- s_i$.

Suppose that t can be defined by a set of k nonrecursive rules. Let r_1 through r_k denote the bodies of these k rules. If we view r_1 through r_k as relational expressions, then the relation for t is the union of the relations for the r_i. Let R denote the set containing the r_i, and S denote the expansion of t.

Consider some string r_j in R. Define a one-one mapping h from the variables of r_j to some set of constants. Then construct a representative database *edb as* follows: if $p(V_1, V_2, \ldots, V_n)$ appears in r_j, add the tuple $(h(V_1), h(V_2), \ldots, h(V_n))$ to the relation for p in *edb*. By the definition of *edb*, if $D_1, \ldots, D_n$ are the distinguished variables, h proves that $(h(D_1), \ldots, h(D_n))$ is in the relation returned by evaluating r_j over *edb*.

Because R and S both define t, there must be some string s in S such that $(h(D_1), \ldots, h(D_n))$ is in the relation returned by evaluating s over *edb*. Thus there is a mapping g from the variables of s to the constants in *edb* such that distinguished variables map to themselves, and the tuples of variables in s map consistently to tuples of constants in *edb*. Since h is one-to-one, it is invertible. Then the composition $g \circ h^{-1}$ is a containment mapping from s to g.

We can repeat this argument for every string in R to prove that every string in R is mapped to by some string in S. (This is the result by Sagiv and Yannakakis [15] concerning equivalences between unions of tableaux.) Because R is finite, there must be some n_0 such that every string in R is mapped to by one of the first n_0 strings of S. Now consider some string s_n, where $n > n_0$. Again using the preceding arguments, we can show that there is some r_j such that there is a containment mapping m_1 from r_j to s_n. But there is also a containment mapping m_2 from some s_i, $i \leq n_0$, to r_j. Then $m_2 \circ m_1$ maps s_i to s_n. This argument holds for any $n > n_0$, so the proof is complete. ∎

We close this section by noting that if we allow the nonrecursive predicates to be IDB predicates, then any necessary condition for data independent recursion must take into account the definitions of these predicates. As an example, if we take the recursive rule of Example 2.1 and add r_3:

r_3: $e(X, Y) :- a(X), b(Y),$

to r_1 and r_2, then t can be defined by the nonrecursive rule

r_1': ***t(X, Y) :- a(X), b(Y).***

and the recursion strongly data independent. In the following, we assume that all nonrecursive predicates are EDB predicates.

3 The A/V Graph

In Section 2, we reduced the question of data independence to the existence of mappings between the strings in the expansion of the recursively defined predicate. The existence of these mappings depends on the patterns of shared variables in the strings. In this section we return to the class of rules handled by procedure ExpandRule — one linear recursive rule with no repeated variables or constants in the rule head.

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 a directed 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.***

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. ∎

The following properties of A/V graphs are immediate from their definition:

1. Each edge in the graph is between an argument node and a variable node.
2. Each distinguished variable node has exactly one incident unification edge; nondistinguished variable nodes have no incident unification edge.

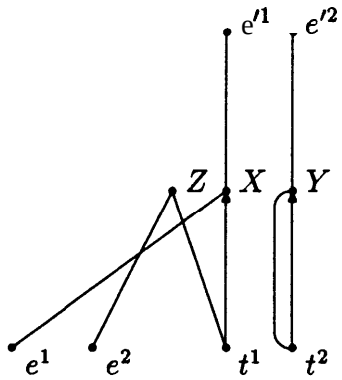

Figure 2: A/V graph for Example 2.1.

3. Each argument node has exactly one incident identity edge; argument nodes for positions of the recursive predicate are also the source of exactly one unification edge.

The A/V graph is similar to the α-graph developed independently by Ioannides. The α-graph differs primarily in that there are no argument nodes; the information about the variables appearing in each predicate is represented by *predicate edges, which,* for every nonrecursive predicate, connect the nodes for the variables appearing in that predicate and are labeled with the name of that predicate. Because the α-graph has no argument nodes, some information is lost.

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 was added to S on iteration i, it can be produced by applying the exit 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 end of a unification edge is replaced by the variable appearing in the argument at the beginning. On iteration 0, because of the initialization of `CurString`, these 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}

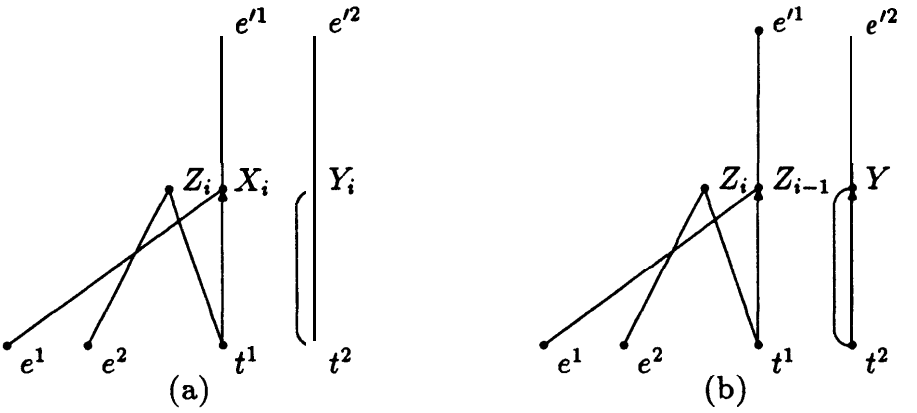

Figure 3: A/V graph for Example 2.1.

and Y. (Figure 3(b)).

After the substitution, argument p of a predicate 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 added by the nonrecursive rule will be $e'(Z_{i-1}, Y)$, and the predicates 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 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 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.) The following two lemmas show that 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.*

Proof: Consider starting at the node for some nondistinguished variable W_0. The only edges incident on nondistinguished variables are identity edges, so if we explore

any path p out of W_0, the first edge must be an identity edge. Suppose this identity edge goes to an argument *al*.

Since a_1 has exactly one incident identity edge, if p can be extended, the second edge in p must be a unification edge, which by definition connects a_1 and some distinguished variable node, say V_2. The edges incident on V_2 are one unification edge, which has already been traversed, and some number of identity edges. Then if we extend p, the next edge must be an identity edge to a_3. Furthermore, since we've already traversed the only identity edge incident on *al*, $a_3 \neq$ *al*.

We prove by induction that for all $j > 0$, edge $2j$ must be a unification edge to a distinguished variable node that hasn't been visited previously, while edge $2j + 1$ must be an identity edge to an argument node that also hasn't been visited previously.

The basis, $j = 1$, has been shown. Suppose that edge $2j - 1$ is an identity edge to an argument node a_{2j-1} that hasn't been visited previously. Because a_{2j-1} has a single incident identity edge, if we extend p, edge $2j$ must be a unification edge to a distinguished variable node V_{2j}. Furthermore, because each distinguished variable node has only one incident unification edge, V_{2j} hasn't been visited previously.

Now consider extending the path from V_{2j}. Edge $2j + 1$ must be an identity edge to a_{2j+1}. But because a_{2j+1} has only one incident identity edge, a_{2j+1} can't have been visited previously. ∎

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

Proof: Suppose that we explore a path p starting at some distinguished variable V_0. V_0 has an incident unification edge; if we follow it, we reach an argument node, say a_1. There must be an identity edge out of *al*; the only way to extend p is to take this identity edge. This edge must go to some distinguished variable V_2, or else the connected component would contain a nondistinguished variable. If $V_2 = V_0$, we have a cycle. If not, we continue with V_2. Because there are only finitely many distinguished variables, we must eventually return to some previously visited distinguished variable node, and there will be 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, and Z is the only nondistinguished variable in the tree. By Fact 3.1, Z_i appears in e^2 and e'^1 on iteration i, and in e^1 on iteration $i+1$. ∎

In Section 4 it will be important to know how variables are shared between the predicates 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 For $i \geq max(j, k)$, a variable appears in position p_1 of a predicate produced on iteration $i+j$, and in position p_2 of a predicate produced on iteration $i+k$, if and only if there is a path from from p_1 to p_2 of weight $k-j$.*

Proof: If V is a nondistinguished variable, then by Fact 3.1, V appears in p_1 if and only if there is a path from the node for V to p_1 containing j unification edges, and a path from the node for V to p_2 containing k unification edges. The concatenation of these two paths is a path from p_1 to p_2 of weight $k-j$.

If V is a distinguished variable, then by Fact 3.2, V appears in p_1 if and only if there is a path from the node for V to p_1 containing $i+j$ unification edges, and a path from the node for V to p_2 containing $i+k$ unification edges. Again, the concatenation of these two paths is a path from p_1 to p_2 of weight $-(i+j)+(i+k) = k-j$. ∎

The proof shows that the path can always be divided into two segments. The first runs from p_1 to some variable node V and contains only identity and reverse unification edges; the second runs from V to p_2 and contains only identity and forward unification edges. The clause "for $i \geq max(j, k)$" is necessary because for $i < max(j, k)$, the argument positions at the ends of the two segments could still contain variables from intermediate nodes in the segments.

Example 3.3 Consider the following pair of rules:

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

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

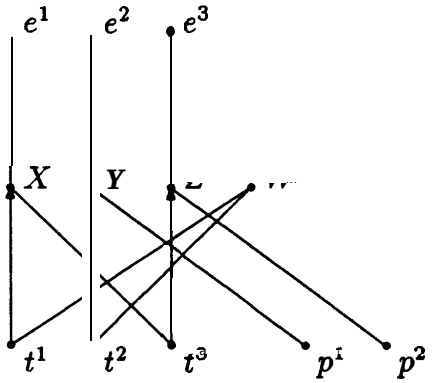

Figure 4: A/V graph for Example 3.3

The first four strings generated are

$$e(X,Y,Z),$$
$$e(W_1,W_1,X)p(Y,Z),$$
$$e(W_2,W_2,W_1)p(W_1,X)p(Y,Z),$$
$$e(W_3,W_3,W_2)p(W_2,W_1)p(W_1,X)p(Y,Z).$$

The A/V graph for these rules is given in Figure 4. There is a path from p^1 to p^2 of weight $(-1)+2=1$, so by Lemma 3.3, with $j=0$, and $k=1$, for $i \geq 1$, position p^1 in a predicate produced on iteration i shares a variable with position p^2 in a predicate produced on iteration $i+1$. ∎

4 Testing for Data Independence

4.1 Unbounded Chains

In Section 2, we reduced the question of data independence to the existence of mappings between the strings in the (infinite) expansion of the recursively defined predicate. The existence of those mappings is in turn tied to the presence of ***unbounded* chains in the** expansion. Informally, the expansion contains unbounded chains if, for any ***n*, *we can* find** a string of the expansion that contains a subsequence of $\geq n$ predicates such that each shares a nondistinguished variable with the next. Strictly speaking, no single string in the expansion contains an unbounded chain. However, we will occasionally refer to chains that grow from string to string as instances of the unbounded chain.

Example 4.1 In the expansion of the transitive closure rules, (Example 2.1), string n contains a sequence of n e predicates, linked by the Z_i's. Thus the expansion contains unbounded chains. ∎

It is easiest to define unbounded chains in terms of the A/V graphs for rules that produce them. Unbounded chains depend only on the recursive rule; to detect them, we augment the A/V graph for the rule by adding predicate edges between adjacent argument positions of each nonrecursive predicate. These predicate edges have weight zero.

Definition 4.1 A path in the the augmented A/V graph for a recursive rule is a ***chain generating path*** if and only if

1. For every argument position p on the path, there is a path containing no predicate edges from some nondistinguished variable node to p, and
2. It is a simple cycle of non-zero weight.

The following lemma justifies the name "chain generating path." In this subsection we will say that each occurrence of a predicate in the body of the recursive rule is a different instance of that predicate.

Lemma 4.1 ***The strings in*** S ***contain a sequence of predicates*** $p_1, p_2, \ldots, p_n = p_1$ ***such that***

1. p_1 ***and*** p_n ***were produced on different iterations by the same predicate instance in the recursive rule, and***
2. ***For*** $1 \leq i < n$, p_i ***and*** p_{i+1} ***share a nondistinguished variable***

if and only if there is a chain generating path in the augmented A/V graph for the recursive rule.

Such a sequence is called a *link*, and the positions holding the shared variables the ***linking positions.*** Because the first and last predicates of a link are instances of the same predicate, produced on different iterations, the last predicate of one link can be the first predicate of another. Thus for any n, we can find a string in the expansion that contains a subsequence of n predicates, each connected to the next by a shared nondistinguished variable.

Proof: (Lemma 4.1) Suppose that there is a chain generating path containing n predicate edges. The predicate edges partition the path into $n-1$ segments. Let the i^{th} segment start in position p_i^2 of predicate p_i, end in position p_{i+1}^1 of predicate p_{i+1}, and have weight k_i. By Lemma 3.3, the variable appearing in p_i^2 on iteration j will appear in position p_{i+1}^1 on iteration $j + k_i$. Thus sufficiently long strings in the expansion will contain sequences

$$p = p_1 \xrightarrow{k_1} p_2 \xrightarrow{k_2} p_3 \cdots p_{n-1} \xrightarrow{k_n} p_n = p,$$

where the label on each arrow is the number of iterations between the appearances of the predicates on either side of the arrow, and the predicates on either side of

an arrow share a nondistinguished variable. Because the total weight of the cycle is nonzero, p_1 and p_n can't have been produced on the same iteration.

Now assume that the strings of the expansion contain a sequence $p_1, p_2, \ldots, p_n = p_1'$ such that p_1 and p_1' were produced on different iterations by the same predicate instance in the recursive rule, and that for $1 \leq i < n$, p_i and p_{i+1} share a nondistinguished variable. We can assume without loss of generality that no three predicates share the same nondistinguished variable. Also, we can assume that with the exception of p_1 and p_1', the p_i where produced by different predicate instances in the recursive rule. Then for $1 \leq i < n$, in p_i there are distinct argument positions p_i^1 and p_i^2 such that p_i^2 and p_{i+1}^1 share nondistinguished variables. By Lemma 3.3, for $1 \leq i < n$ there must be a path in the A/V graph from p_i^2 to p_{i+1}^1 of weight k_i. These paths are connected by predicate edges, and form a simple cycle. We claim that this cycle is a chain generating path.

By Fact 3.1, a nondistinguished variable W_j appears in position p on iteration i only if there is a path from W to p of weight $i - j$. Then because the linking positions of the sequence contain nondistinguished variables, for all p on the path there must be a predicate edge-free path from some nondistinguished variable node to the node for p.

Since p_1 and p_1' were produced on different iterations, $\sum_1^{n-1} k_i \neq 0$, and the cycle will have a nonzero weight, so the path is a chain generating path. ∎

Example 4.2 The chain generating paths in most commonly exhibited recursive rules consist of a single segment. In the transitive closure example (Example 2.1 and Figure 5), the chain generating path visits e^1, e^2, Z, t^1, and X. The corresponding link is a pair of e predicates, produced on iteration i and $i + 1$ for all i, and the corresponding unbounded chain begins

$$e(X, Z_0)e(Z_0, Z_1)e(Z_1, Z_2)e(Z_2, Z_3)\ldots$$

For a two segment chain generating path, consider the following recursive rule:

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

The A/V graph for this rule is given in Figure 6. One segment of the chain generating path goes from p^2 to q^1; the other goes from q^2 to p^1. The corresponding link is

$$p(Z_i, W_{i+1})q(W_{i+1}, Z_{i+1})p(Z_{i+1}, W_{i+2}),$$

and the corresponding unbounded chain begins

$$p(X, W_0)q(W_0, Z_0)p(Z_0, W_1)q(W_1, Z_1)p(Z_1, W_1)q(W_1, Z_2)\ldots$$

∎

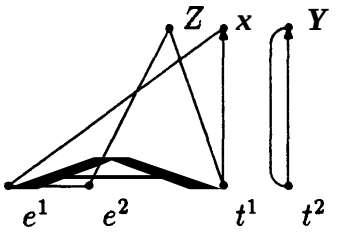

Figure 5: Augmented A/V graph for Example 1.1.

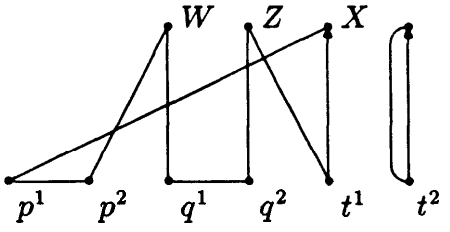

Figure 6: Augmented A/V graph for the two segment rule in Example 4.2.

There are two more important facts about unbounded chains.

Fact 4.1 In the chains produced by a chain generating path, no variable appears twice in any linking position.

Fact 4.2 In the chains produced by a chain generating path, there is a distinguished variable in at least one linking position in the first predicate of the chain.

Proof: (Fact 4.1) By Fact 3.1, since there are paths from nondistinguished variables to all positions on the chain generating path, eventually all positions on the chain generating path will contain nondistinguished variables. We prove Fact 4.1 by showing that if a nondistinguished variable appears in position p, no variable appears twice in position p. Suppose that W_i appears in position p on iterations $i + k_1$ and $i + k_2$. Then there must be paths from W to p of lengths k_1 and k_2. But then, again by Fact 3.1, if $k_1 < k_2$, on iteration $i + k_2$, both W_i and $W_{i+k_2-k_1}$ must appear in p.

(Fact 4.2) Suppose we arbitrarily pick some node and make one complete cycle of the chain generating path. Let w_{min} and w_{max} be the minimum (maximum) weight of any node on the cycle, relative to the starting node. Then if $k = w_{max} - w_{min}$, the predicates produced on the first $k + 1$ iterations will contain one complete link. Let this link be $p_1, p_2, \ldots, p_n = p_1$. We will continue to use p_1 and p_n to distinguish the two occurrences of the same predicate.

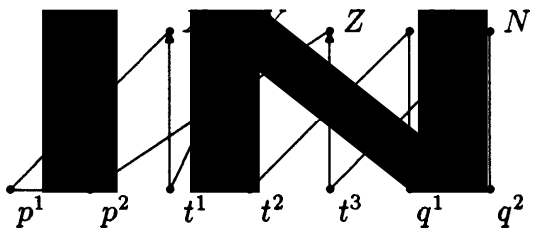

Figure 7: Augmented A/V graph for Example 4.3.

p_n in its role as the last predicate in the first link, must share some variable with a predicate in the second link of the chain, say q. By Lemma 3.3, there must be a path from an argument position of q to an argument position of p. However, p_1 cannot share a variable with an earlier instance of q, or else the sequence of predicates from this instance of q to the later instance of q would be the first link of the chain. This is possible only if p_1 was produced on iteration i, and the path from the variable node for the variable shared between p and q to the position of p has more than i reverse unification edges. But then on iteration i, this position of p must contain a distinguished variable. ∎

Example 4.3 The A/V graph for the following rule (Figure 7) has a chain generating path that illustrates some of the points in the proof of Fact 4.2.

$$t(X,Y,Z) :- p(X,Z), t(Y,M,N), q(M,N).$$

There is a two-segment chain generating path, with one segment from q^1 to p^1, and the other from p^2 to q^2. If we start at position q^1 and traverse the path, we find that $w_{max} - w_{min} = 2 - 0 = 2$, so a complete link will be generated in the first three iterations. The predicates produced on the first three iterations, subscripted by the iteration on which they appeared, are

$$p_2(N_0, M_1)q_2(M_2, N_2)p_1(Y, M_0)q_1(M_1, N_1)p_0(X, Z)q_0(M_0, N_0)$$

The first link runs from p_1 to q_0 to p_2. p_2^1 shares N_0 with q_0^2, and there is a path containing 2 unification edges from N to p^1. Because p_1 is produced on iteration 1, p_1^1 shares no variable with any q instance, and contains the distinguished variable Y. ∎

4.2 Strong Data Independence

Strong data independence is equivalent to data independence with an arbitrary initialization of the recursive relation. It is easier to test for than weak data independence because there is no possibility of a recursive rule that would be data dependent but for some nonobvious interaction with the nonrecursive rule.

Theorem 4.1 ***A linear recursive rule is strongly data independent if there is no chain generating path in the augmented A/V graph for the rule.***

Proof: By showing a stronger condition: if the A/V graph for a rule contains no chain generating path, then there is a k such that if a string s is generated by more than k successive applications of the rule, then s is mapped to by a string generated by no more than k successive rule applications.

We first prove a bound on the distance between occurrences of the same nondistinguished variable in the strings of the expansion.

Let l be the maximum weight of any acyclic path in the A/V graph. By Lemma 3.1, no nondistinguished variable can appear in a position reachable from a cyclic path. Then Lemma 3.3 implies that no nondistinguished variable appears in two predicates produced on iterations more than l apart.

By Lemma 4.1, if there is no chain generating path, there is no sequence of predicates, linked by shared nondistinguished variables, that starts and ends in two instances of the same predicate. Let n be the number of predicates in the recursive rule. Then in any substring produced by more than ln consecutive applications of the recursive rule, there is no predicate connected, via a sequence of predicates sharing nondistinguished variables, to predicates appearing both before and after the substring. (If there were, there would be a sequence of predicates satisfying the conditions of Lemma 4.1, which would imply a chain generating path.)

Let $L = ln^2$, the number of predicates produced by ln consecutive applications of the rule, and let I be the number of nonisomorphic instances of strings of L predicates. Furthermore, let r be the number of nonisomorphic instances of the recursive predicate t. Then any string s that is longer than rIL can be written as

$$s = \ldots p_1 \alpha p_2 \beta p_1' \alpha' p_2' \ldots,$$

where $p_1 \alpha p_2$ and $p_1' \alpha' p_2'$ are isomorphic, each is L predicates long, and the instances that generated p_2 and p_2' are isomorphic. Here p_1 and p_2 are predicates, and α and β are strings of predicates.

Consider the string s',

$$s' = \ldots p_1 \alpha p_2' \ldots,$$

formed by deleting the predicates in s from p_2 up to but not including p_2'. Because the instances of that generated p_2 and p_2' were isomorphic, s' can be generated by deleting the subsequence of rule applications that produced $p_2 \beta p_1' \alpha'$ from the sequence of rule applications that produced s. Thus s' is also in S.

We claim that s' maps to s. No predicate appearing in α in s' can be linked by predicates sharing nondistinguished variables to predicates appearing both before p_1 and after p_2', and any distinguished variables in α appear in the same positions as they do in α and in α' in s. If a predicate in α in s' is linked to predicates

appearing before p_1, map it to the corresponding instance in α in s; else, map it to the corresponding instance in α'. Map all predicates appearing after p_2' in s' to the corresponding predicates appearing after p_2' in s, and map all predicates appearing before p_1 in s' to the corresponding predicates appearing before p_1 in s. ∎

Chain generating paths can be detected in time linear in the length of the recursive rule. The algorithm has two phases, each phase checking one part of the definition.

Phase 1 uses depth-first search on the non-augmented A/V graph to discover the connected components for each distinguished variable node, and removes all nodes in any connected component that contains a cycle. The argument nodes in such a component are exactly the argument nodes that will always contain distinguished variables; removing them leaves exactly the nodes that can be reached from a nondistinguished variable node.

The second phase operates on the augmented A/V graph, restricted to the nodes that survived phase 1. Phase 2 uses depth-first search, starting at each nondistinguished variable node W, to search for a node that can be reached by two paths with different weights. The A/V graph contains a chain generating path if and only if there is such a node. (Phase 2 is essentially Algorithm 6.1 in Ioannides [7].)

Unfortunately, a chain generating path is not a sufficient condition for data dependent recursion. The strongly data independent rules with chain generating paths are strange.

Example 4.4

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

is a strongly data independent rule with a chain generating path. ∎

It is easy to define subclasses of rules such that a chain generating path is indeed a sufficient condition for data dependence. One such class, rules for which the (non-augmented) A/V graph contains no cycles, was considered by Ioannides [7]. Another such class is rules with no repeated nonrecursive predicates. To prove this, we use another fact about unbounded chains.

Fact 4.3 Let r be a linear recursive rule with no repeated nonrecursive predicates, and let the expansion for r contain unbounded chains. Then if string s_1 in the expansion maps to another string s_2, all variables in the unbounded chain in s_1 must map to themselves in s_2.

Proof: By induction on the positions of the predicates in which the variables appear.

Let p be the first predicate in the chain. By Fact 4.2, there is a position a in p that contains a distinguished variable. Then, because there are no repeated predicates in the rule, by Fact 4.1, this is the only instance of p such that a contains that distinguished variable. Thus if this instance of p is to map any predicate in s_2, every variable in this instance of p must map to itself.

Assume that the variables in predicate i in the chain in s_1 must map to themselves in s_2. Predicate $i+1$ shares a linking variable, say W, with predicate i, so W must map to itself. Suppose W appears in position b in predicate $i+1$. As there are no repeated predicates in the rule, Fact 4.1 implies that predicate $i+1$ is the only instance of the predicate that contains W in position b. Thus every variable in predicate $i+1$ in s_1 must map to itself in s_2. ∎

Theorem 4.2 ***A linear recursive rule with no repeated nonrecursive predicatea is strongly data independent*** *if* ***and only if there is no chain generating path in the augmented A/V graph*** *for* ***the rule.***

Proof= The "if" part is given by Theorem 4.1.

For the "only if" part, suppose that there is a chain generating path in the A/V graph for the rule, and that the recursively defined predicate is $t(X_1, \ldots, X_m)$. We show that if we add the base rule

r_t: $t(X_1, \ldots, X_m) :- t_0(X_1, \ldots, X_m)$.

where t_0 doesn't appear anywhere in the recursive rule, then the recursion is data dependent.

In any chain generating path, there must be at least one segment that is of positive weight. Suppose that this segment runs from position p to position p'. Since this path must contain a unification edge, it must pass through a distinguished variable node. But every distinguished variable node shares an identity edge with some position of t_0, so there must be a positive weight path from p to some position of t_0. Then by Lemma 3.3, t_0 shares a linking variable with the unbounded chain.

Let W_m be a linking variable shared between the chain and t_0 in s_1, and suppose that W_m appears in argument position t_0^i. By Fact 4.3, W_m must map to itself in s_2. But because s_2 is longer than s_1, the variable appearing in t_0^i in s_2 can't be W_m. Because there is only one instance of t_0 in the string, s_1 can't map to s_2, and by Theorem 2.1, the recursion is data dependent. ∎

Example 4.5 The augmented A/V graphs for the recursive rules of Examples 2.1, 2.4, and 2.5 all satisfy the conditions of Theorem 4.2, so they are not strongly data independent. The augmented A/V graph for the rule

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

is given in Figure 8. There is no chain generating path, so the rule is strongly data independent. ∎

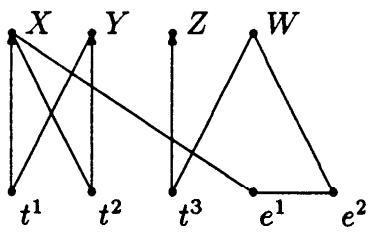

Figure 8: Augmented A/V graph for Example 4.5

4.3 Weak Data Independence

If we pair a strongly data independent rule with any exit rule, the pair will be data independent. However, the recursive rule in a weakly data independent recursive rule-exit rule pair need not be strongly data independent. Results for weak data independence are less clean than those for strong data independence because they must deal explicitly with interactions between the recursive rule and the exit rule.

Example 4.6 Consider again the transitive closure rule,

r_1: *t(X, Y) :- e(X, Z), t(Z, Y)*

where e is an EDB predicate. There is a chain generating path in the A/V graph for this rule, so it is not strongly data independent. If we add the usual nonrecursive rule

r_2: *t(X, Y) :- e(X, Y),*

the recursion is data dependent. However, if we replace r_2 with

r_2': *t(X, Y) :- e(W, Y).*

then t is completely defined by the exit rule r_2'. Our second example shows what can happen if there are multiple nonrecursive predicates in the recursive rule.

r_3: $t(X,Y) :- t(X,Z), e(Z,Y), e(X,W), e(W,Y).$

r_4: *t(X,Y) :- e(X,Y).*

Here, after the first string, all strings contain the two-predicate chain $e(X, W_1)e(W_1, Y)$, so the second string in S, $e(X, Z_1)e(Z_1, Y)e(X, W_1)e(W_1, Y)$, maps to all subsequent strings, and the recursion is data independent. ∎

Recently, Vardi [19] has proven that weak data independence is undecidable even for recursive definitions containing only one linear recursive rule. However, we can test for necessary and sufficient conditions for weak data independence in predicates defined by

1. One ***regular*** recursive rule (a regular rule is a linear recursive rule with a single nonrecursive predicate), and

2. One nonrecursive rule with a single-predicate body.

Proofs of necessary and sufficient conditions for weak data independence for such rules closely parallel the proof of Theorem 4.2. In that proof we paired the recursive rule with a nonrecursive rule with a single predicate body, t_0. We used two properties of t_0. First, in the strings of the expansion, t_0 shares a nondistinguished variable with the unbounded chain. Second, if s_1 is to map to s_2, the t_0 instance in s_1 must map to the t_0 instance in s_2.

If the predicate of the nonrecursive rule satisfies the first property, we say that it is *connected* to the unbounded chain; if it satisfies the second, *we* say it is *irredundant.* When considering strong data independence, we can choose the predicate in the nonrecursive rule so that it is connected and irredundant. The task in weak data independence is to decide, given a recursive-nonrecursive rule pair, whether the predicate in the nonrecursive rule has these properties. If it does, then we can apply the following theorem:

Theorem 4.3 *Given a regular recursive-nonrecursive rule pair, with e being the predicate in the nonrecursive rule body, the recursion is data dependent if and only if*

1. *There is a chain generating path in the augmented A/V graph for the recursive rule, and*
2. *e is connected to the unbounded chain, and*
3. *e is irredundant.*

Proof: The "if" condition follows exactly that of Theorem 4.2.

The "only if" part uses a cyclic property of the strings of S. Since there are only a finite number of nonisomorphic instances of t, if we observe the instances of t in `CurString` through successive iterations, we must eventually see two that are isomorphic. Furthermore, after a startup interval of I iterations, if the number of iterations between the isomorphic instances is τ, then for any i, the instances of t in `CurString` on iteration i and on iteration $i+\tau$ are isomorphic. (The startup interval is the number of iterations until distinguished variables that appear in no cycles have disappeared, and is bounded above by the maximum number of unification edges in any acyclic path in the A/V graph.) Since the predicates added to `CurString` on any iteration depend only on the recursive rule (which is fixed) and on the instance of t, for any m, a predicate produced on iteration $I+i$ is isomorphic to one produced on iteration $I + m\tau + i$.

By Theorem 4.1, a chain generating path is a necessary condition for data dependent recursion. Suppose that there is a chain generating path, but that e is not connected to the unbounded chain. Then the only variables e can share with the unbounded chain are distinguished.

Because predicates produced $m\tau$ iterations apart are isomorphic, distinguished variables appear in the same positions in predicate instances produced $m\tau$ iterations

apart. Then for all m and j, the string

$$p_1 \cdots p_{I+1} \cdots p_{I+\tau} p_{I+\tau+1} \cdots p_{I+2\tau} \cdots p_{I+m\tau} \cdots p_{I+m\tau+j} e$$

is mapped to by

$$p_1 \cdots p_{I+1} \ldots p_{I+\tau} p_{I+\tau+1} \ldots p_{I+\tau+j} e,$$

where we map p_i in the second string to p_i in the first, and the e predicate to itself.

Finally, if e is redundant, by definition of irredundance, we can map s_1 to s_2 by mapping all predicates but e to themselves, and mapping e to some predicate in s_2. ∎

We can decide both irredundance and connectedness from the A/V graph for the pair of rules.

Definition 4.2 The predicate e of the exit rule, is *irredundant* if either

1. $e \neq p$, or
2. There is a path from a distinguished variable node V, where V is on a cycle, to an argument of e such that there is no path from V to the same argument of p, or
3. There are paths from some variable node V to two distinct arguments of e, each of weight k, and no j such that there are paths of weight j from some variable node to both of the corresponding arguments of p, or
4. Let $\{V_1, \ldots, V_n\}$ be the distinguished variable nodes such that there is an identity edge from V_i to e^{j_i}, and for each V_i there is a positive weight path from some argument of p to V_i. Then there must be no k such that, for all i, there is a path of weight k from V_i to p^{j_i}.

This definition is complex, but the intuition is simple. Either of conditions one, two, or three guarantee that the exit predicate maps to no other predicate in any string. In condition four, the variables at the beginnings of paths to the V_i are the variables that are shared between the exit predicate and the chain. By Fact 4.3, if one string is to map to another, these variables must map to themselves. If there is no k as described, then e will contain these variables in a pattern different from any p predicate in any string, and again e must map to itself.

Definition 4.3 The predicate e of the nonrecursive rule is *connected* to the unbounded chain if and only if there is a positive weight path from some argument of p, through some nondistinguished variable node, to an argument of e.

By Lemma 3.3, this definition guarantees that e will share a nondistinguished variable with some instance of p in the chain.

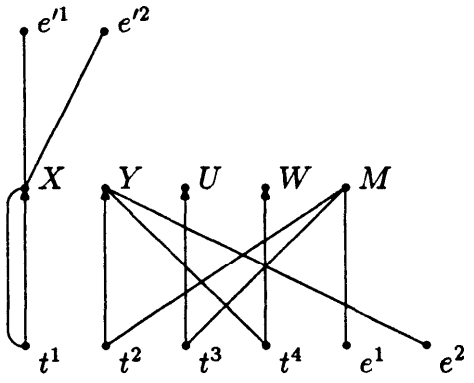

Figure 9: A/V graph for rules r_r and r_e in Example 4.7.

Example 4.7 Consider the following rules:

r_r: *t(X, Y, U, W) :- t(X, M, M, Y), e(M, Y).*

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

Since the nonrecursive predicate in both rules is e, we use e' for the instance of e in r_e. In the A/V graph for these rules (Figure 9), there is no path satisfying Definition 4.3, and the exit predicate is not connected. The expansion for t begins

$$e'(X,X),$$
$$e'(X,X)e(M_0,Y),$$
$$e'(X,X)e(M_1,M_0)e(M_0,Y),$$
$$e'(X,X)e(M_2,M_1)e(M_1,M_0)e(M_0,Y).$$

Any string in the expansion can be mapped to any subsequent string.

If we replace r_e by

r'_e: *t(X, Y, U, W) :- e(U, W).*

then the path in Figure 10 from e^1 to e'^1 proves that e' is connected. However, condition 4 of Definition 4.2 is not satisfied. The distinguished variable nodes with identity edges to nodes of e' are U and W, and there are positive weight paths from arguments of e to both U and W. Also, there are paths containing one unification edge from U to e^1, and from W to e^2. This implies that a string produced on iteration i will map to all longer strings by mapping the e predicates to themselves, and mapping e' to the instance of e produced on iteration $i-1$. The first four strings of the expansion confirm this:

$$e'(U,W),$$
$$e'(M_0,Y)e(M_0,Y),$$
$$e'(M_1,M_0)e(M_1,M_0)e(M_0,Y),$$
$$e'(M_2,M_1)e(M_2,M_1)e(M_1,M_0)e(M_0,Y).$$

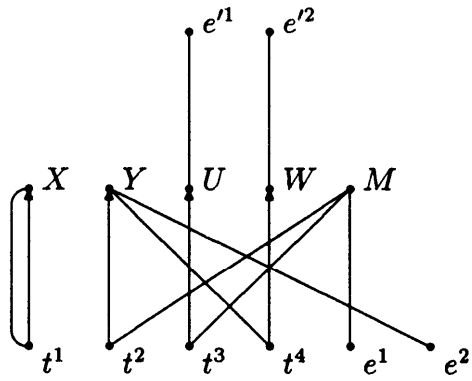

Figure 10: A/V graph for rules r_r and r'_e of Example 4.7

If we replace r_e by

r''_e: $t(X, Y, U, W)$:- $e(U, U)$.

then the paths from U to e'^1 and e'^2 in Figure 11 satisfy condition 3 of Definition 4.2, and exit predicate is no longer redundant. Theorem 4.3 says that the recursion is data dependent. Here the first four strings are

$$e'(U,U),$$
$$e'(M_0,M_0)e(M_0,Y),$$
$$e'(M_1,M_1)e(M_1,M_0)e(M_0,Y),$$
$$e'(M_2,M_2)e(M_2,M_1)e(M_1,M_0)e(M_0,Y).$$

∎

5 An Extension to Multiple Rules

Consider a predicate defined by n linear recursive rules t:- $t_i, p_{i1}, \ldots, p_{ik_i}$, where $1 \leq i \leq n$, and m nonrecursive rules t :- $e_{j1}, \ldots, e_{jk_j}$, where $1 \leq j \leq m$. The subscripts on the t predicates are to distinguish the instances in different rules, while the p_{ik} and e_{ij} may be distinct sets of predicates.

Gaifman [5] has shown that deciding if such a set of rules is weakly data independent is undecidable, and Mairson and Sagiv [9] have extended his result to show that even strong data independence is undecidable for multiple linear recursive rules. However, we can extend Section 4 to give a sufficient condition for strong data independence in sets of recursive rules.

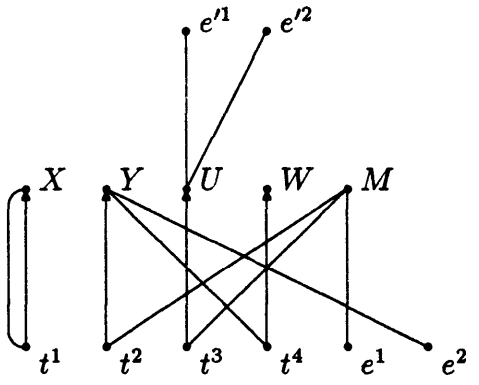

Figure 11: A/V graph for rules r_r and r''_e of Example 4.7

Procedure `ExpandRule` must be changed so that `CurString`, instead of being a single string, is a growing set of strings. The procedure methodically applies the rules in all possible ways so that on iteration *i*, all strings that can be generated by i applications of the rules are produced.

We can represent the set `CurString` over time as a tree, where the label of the root is t, and the labels of nodes on level i are the strings that were the elements of `CurString` on iteration i. Paths in this tree correspond to sequences of rule applications — a child node is the result of one more rule application to the sequence that produced its parent. Each node in this tree has associated with it the *m* strings of base predicates generated by applying the *m* exit rules to its label. From this perspective, the above procedure is a breadth-first construction of the ***rule/goal tree*** [17] for the input rules.

We must also extend the interpretation of the A/V graph. In a multiple rule A/V graph, in addition to telling where a variable appears in the strings of S, paths specify sequences of rule applications. Informally, when following a path, we start by assuming that a variable *V* appears in some position *p* on an iteration i. This implies that the rule containing *p* was applied on iteration i. As we follow paths through the A/V graph, taking identity edges corresponds to moving between argument positions of predicates produced on the same iteration. Taking a unification edge in the forward direction specifies the rule to be applied on iteration $i+1$, while taking a unification edge in reverse specifies the rule to be applied on iteration $i-1$. By induction, if we start by considering what variable appears in a position *p* on iteration i, and the weight of the path traversed from *p* to a position *p'* of rule r_j is w, then rule r_j must be applied on iteration $i +$ w.

There are some paths that imply that a string was generated by applying multiple rules on a single iteration. Such paths are termed inconsistent and must be disallowed. The relationship between path weights and iterations suggests the following definition.

Definition 5.1 A path through an A/V graph is ***inconsistent*** if it contains argument

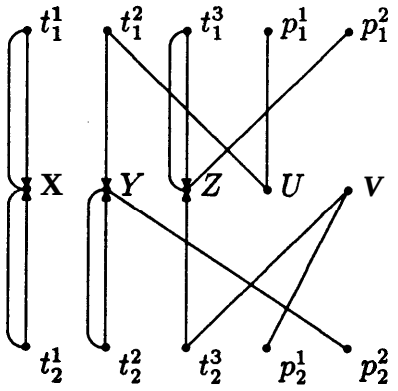

Figure 12: A/V Graph for the recursive rules of Example 5.1

positions p and p 'such that p and p 'appear in different rules, and the weight of the prefix of the path to p is the same as the weight of the prefix of the path to p'. A path is *consistent* only if it is not inconsistent.

Example 5.1 Consider the following three rules defining a relation t.

r_1: $t(X,Y,Z) :- t_1(X,U,Z), p_1(U,Z).$
r_2: $t(X,Y,Z) :- t_2(X,Y,V), p_2(V,Y).$
r_3: $t(X, Y, Z) :- e(X, Y).$

A portion of the first three levels of the rule/goal tree for these rules is given in Figure 13. The A/V graph for the same rules is given in Figure 12. The t_2^i and p_2^i are argument positions of rule r_2, and the t_1^i and p_1^i are argument positions of rule r_1. The longest path in Figure 13 corresponds to the sequence of rule applications r_1, r_2, r_1. The string produced by this sequence followed by r_3 is

$$e(X, U_2)p_1(U_2,V_1)p_2(V_1, U_0)p_1(U_0, Z).$$

An inconsistent path through the A/V graph is given in Figure 14. It would require that rule r_1 and rule r_2 both be applied on the same iteration. ∎

In multiple recursive rule A/V graphs, modified versions of Facts 3.1 and 3.2 still hold.

Fact 5.1 There is a sequence of rule applications such that a nondistinguished variable W_i appears in position p in a predicate produced on iteration $i + k$ if and only if there is a consistent path from W to p containing k unification edges.

Fact 5.2 There is a sequence of rule applications such that a distinguished variable V appears in position p on iteration i if and only if there is a consistent path from V to p containing i unification edges.

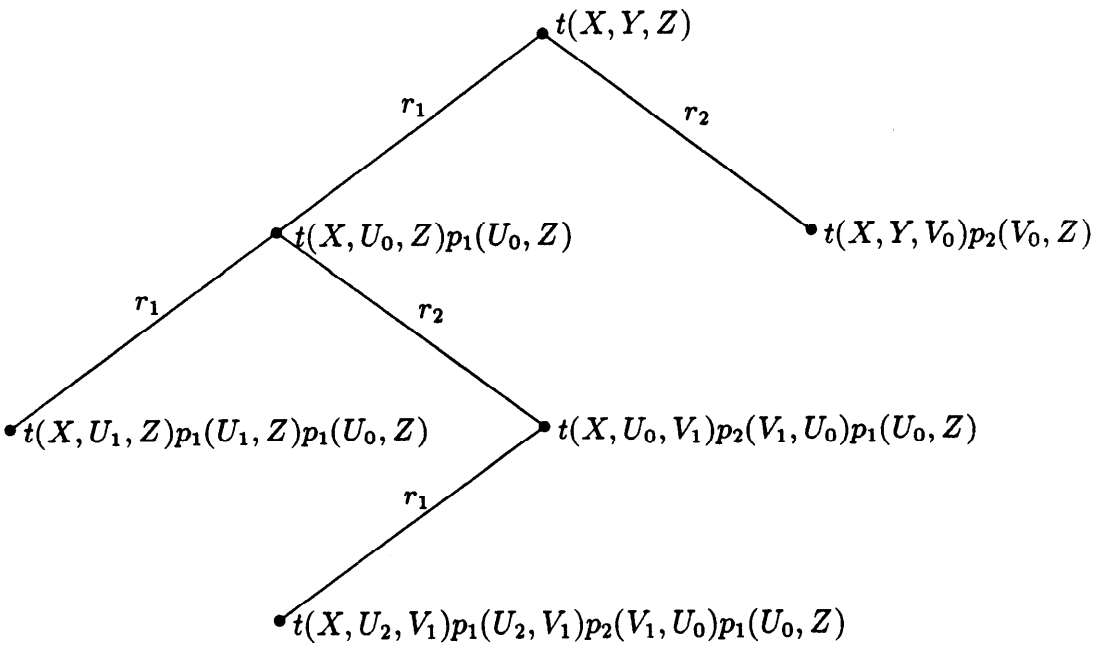

Figure 13: The rule/goal graph for Example 5.1

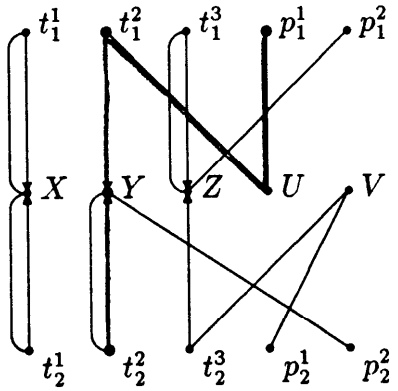

Figure 14: An inconsistent path for the recursive rules of Example 5.1

The following version of Lemma 3.3 holds:

Lemma 5.1 *For $i \geq max(j,k)$, there is a sequence of rule applications such that a variable appears in position p_1 of a predicate produced on iteration $i + j$, and in position p_2 of a predicate produced on iteration $i + k$, if and only if there is a consistent path from p_1 to p_2 of weight $k - j$.*

Proof: Follows closely that of Lemma 3.3, using Facts 5.1 and 5.2 instead of 3.1 and 3.2. ∎

The definition of a chain generating path must be extended in two ways. First, the path must be consistent. Second, it is not enough that there be some path from a nondistinguished variable into every argument position of the chain generating path — these paths must specify sequences of rule applications that are consistent with that specified by the chain generating path. Here we use the fact that the sequence of rule applications specified by a chain generating path can be repeated to generate arbitrarily long chains. Thus if $r_{i_1} r_{i_2} \ldots r_{i_k}$ is the sequence of rules applied along the chain generating path, and $r_{j_1} r_{j_2} \ldots r_{j_m}$ is a sequence of rules applied along a path from a nondistinguished variable to an argument position on the chain generating path, for some n, $r_{j_1} r_{j_2} \ldots r_{j_m}$ must be a substring of $(r_{i_1} r_{i_2} \ldots r_{i_k})^n$. (The sequence of rules is completely determined by the chain generating path, so any two sequences that are consistent with the chain generating path must be mutually consistent.)

We summarize the above two points with the multiple rule definition of a chain generating path.

Definition 5.2 A path in the augmented A/V graph for a set of linear recursive rules is a ***chain generating path*** if and only if

1. It is a simple cycle of nonzero weight, and
2. It is consistent, and
3. For every argument position p on the cycle, there is a path, containing no predicate edges and consistent with the chain generating path, from some nondistinguished variable node to p.

Example 5.2 The darkened path in Figure 15 is a chain generating path. ∎

Fact 4.1, Fact 4.2, and Lemma 4.1 are unchanged.

Theorem 5.1 *A set of linear recursive rules is strongly data independent if there is no chain generating path in the augmented A/V graph for the rules.*

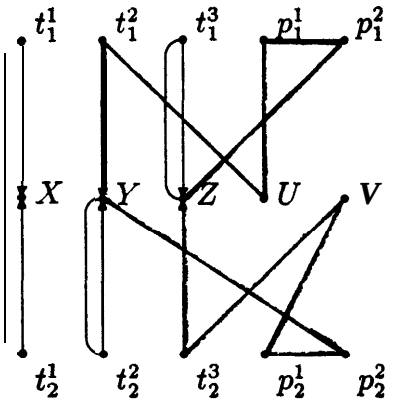

Figure 15: Augmented A/V Graph for the recursive rules of Example 5.1

Proof: By showing a stronger condition: for any n rules, if there exists no chain generating path in the graph for the rules, then there is a constant k such that any string produced by more than k consecutive applications of the rules is mapped to by a string produced by fewer than k applications. The proof is by induction on the number of rules. The basis, one rule, is given by Theorem 4.1.

If we have n recursive rules and no chain generating path, by induction we can find a k_{n-1} such that if we remove any rule, any string produced by more than k_{n-1} applications of the remaining rules is mapped to by a shorter string. Assume that every rule is applied at least once every k_{n-1} rule applications.

If there is no path from a nondistinguished variable node to a cycle, then the largest number of iterations separating two appearances of a nondistinguished variable is bounded above by the maximum number of unification edges in any path from a nondistinguished variable node to an argument node. If there is a path from a nondistinguished variable node of some rule into a cycle, then a new variable is injected into the cycle every time that rule is applied. Since each rule must be applied at least every k_{n-1} rule applications, a variable can appear in predicates produced on iterations at most $D = f_1 + l * k_{n-1} + f_2$ apart, where l is the number of unification edges in the cycle and f_1 and f_2 are the maximum numbers of unification edges in any acyclic paths into and out of the cycle.

Because there is no chain generating path, there can be no subsequence of predicates linked by shared variables such that the first and last were produced by the same predicate instance of the same rule. Thus if D is the maximum number of iterations between appearances of a nondistinguished variable, and there are at most m predicates in any recursive rule, the maximum number of predicates between two predicates linked by a sequence of predicates sharing nondistinguished variables is bounded above by $L = m^2 * n * D$.

From here on the proof follows that of Theorem 4.1. If I is the number of

equivalence classes of isomorphic strings of L predicates, and r is the number of nonisomorphic instances of the recursive predicate R, any string containing more than rIL predicates must contain two isomorphic substrings of length L such that the last predicate of each was generated by isomorphic R instances. We can delete the predicates between the two, and the resulting shorter string is in S and maps to the longer. ∎

We note that it is possible for two strongly data independent rules to combine to form a set that is not strongly data independent. In Example 5.1, the A/V graphs for the individual recursive rules contain no chain generating paths, so by Theorem 4.1 they are strongly data independent. However, in the augmented A/V graph for the pair of rules (Figure 12), there is a chain generating path, so by Theorem 5.1 the pair is not strongly data independent.

6 Applications

Typically, for termination, evaluation algorithms for recursive rules rely either upon assumptions about acyclicity in the base relations or upon expensive duplicate detection tests. If the recursion is recognized as data independent, the recursion can be replaced by the equivalent set of conjunctive relational queries, and can be optimized by standard techniques. We now turn to the data dependent case.

A relation defined by a linear recursive rule can be constructed by evaluating successive strings in the expansion of the rule until some string returns no new tuples. This method would be hopelessly inefficient, and recently proposed algorithms [3,6] improve on this method in two ways. First, they use partial results from one string in the evaluation of the next string. Second, they use constants from the queries that cause the recursive relation to be constructed to restrict lookups during evaluation. Here we present a different kind of optimization, finding predicates that need only be evaluated a bounded number of times per string.

Example 6.1 Consider the following rules:

r_1 ***t(X, Y) :- e(X, Z), b(W,Y), t(Z, Y).***
r_2 $t(X, Y)$:- $t_0(X, Y)$.

Here are the first four strings generated by procedure ExpandRule:

$$t_0(X,Y),$$
$$e(X,Z_0)b(W_0,Y)t_0(Z_0,Y),$$
$$e(X,Z_0)b(W_0,Y)e(Z_0,Z_1)b(W_1,Y)t_0(Z_1,Y),$$
$$e(X,Z_0)b(W_0,Y)e(Z_0,Z_1)b(W_1,Y)e(Z_1,Z_2)b(W_2,Y)t_0(Z_2,Y)$$

The ***b*** predicates need only be evaluated once per string. ∎

The techniques of Section **4** can be used to identify these predicates.

Definition 6.1 A predicate *p* is ***connected*** to an unbounded chain if it shares a nondistinguished variable with a predicate on a chain generating path, or if it shares a nondistinguished variable with a predicate that is connected to an unbounded chain.

Theorem 6.1 *If* ***a predicate p is not connected to an unbounded chain, then there is a constant k such that all but k occurrences of p can be removed from any string in the expansion of the rule.***

Predicates that are not connected to any unbounded chain can be detected in linear time by an extension to the algorithm mentioned at the end of Section 4. An optimizer can then transform the original recursive rule-exit rule pair to a new set of rules, containing one recursive rule, where all remaining nonrecursive predicates in the recursive rule are connected to unbounded chains. The details of this transformation, and a proof of Theorem **6.1,** are given in [11].

We cannot hope to find a complete algorithm that detects data independent recursion in arbitrary sets of rules — the problem is undecidable. However, testing for chain generating paths and removing predicates from the recursive rule, as suggested by Theorem **6.1,** may be a useful part of a query planning process. Although we expect that the majority of recursive rules in an actual system will be data dependent, in many cases not all the predicates need to be evaluated at every level of the recursion. In these cases the avoided redundancy during evaluation should more than pay for the added complexity during planning.

Acknowledgement I'd like to thank Jeff Ullman and Yehoshua Sagiv for their many useful comments on this work.

References

[1] Alfred V. Aho, Ye loshua 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 Databases 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. Optimizing function-free recursive inference rules. Stanford Tech Report, to appear.

[12] Raymond Reiter. Deductive question-answering on relational databases. In H. Gallaire and J. Minker, editors, ***Logic and Databases,*** pages 149–177, Plenum Press, New York, 1978.

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

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

[15] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. ***JACM,*** 27(4):633–655, October 1980.

[16] David E. Smith, Michael R. Genesereth, and Matthew L. Ginsberg. ***Controlling Recursive Inference.*** Technical Report HPP 84-6, Stanford, June 1985.

[17] Jeffrey D. Ullman. Implementation of logical query languages for databases. ***TODS,*** 10(4):289–321, September 1985.

[18] Allen Van Gelder. ***A Message Passing Framework for Logical Query Evaluation. Tech***nical Report STAN–CS–85–1088, Stanford University, 1985.

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