Classification and Compilation of Linear Recursive Queries in Deductive Databases

Cheong Youn, Hyoung-Joo Kim, *Member, IEEE*, Lawrence J. Henschen, and Jiawei Han

***Abstract*— In this paper, we present a graph model which is powerful in classifying and compiling linear recursive formulas in deductive databases. The graph model consists of two kinds of graphs: *I-graph* and *Resolution Graph*. We can extract essential properties of a recursive formula from its I-graph and can easily figure out the compiled formula and the query evaluation plan of the recursive formula from its resolution graph.**

We demonstrate that based on the graph model all the linear recursive formulas can be classified into a taxonomy of classes and each class shares some common characteristics in query compilation and query processing. The compiled formulas and the corresponding query evaluation plans can be derived based on the study of the compilation of each class.

***Index Terms*— Deductive database, graph model, knowledge base, linear recursive query, query evaluation plan, recursive query compilation.**

I. Introduction

LOGIC provides a convenient formalism for studying relational database problems as the semantics of relational databases rely upon the first-order logic. Logic also has been used as an inference system and as a representation language in Artificial Intelligence applications. The major advantage of the logic paradigm is in its declarative semantics in that a user can represent programs declaratively without concern about control flow during execution.

Deductive databases are more toward the logic paradigm than relational databases in the sense that the user can specify queries in a declarative manner and an inferencing mechanism is necessary in processing queries of deductive databases. Deductive databases have been a great research topic in the past decade because of the two converging phenomena [1]:

- the desire to merge database technology and artificial intelligence technology, i.e., to extend database systems to provide them with the functionality of expert systems (namely, inference mechanism) thus creating "Knowledge Base Systems"
- the desire to merge logic programming technology and database technology, i.e., to extend the power of the data language of the database system to that of a general purpose programming language.

Manuscript received February 15, 1989; revised October 16, 1991.

C. Youn is with Bell Communications Research, Piscataway, NJ 08854.

H.-J. Kim is with the Department of Computer Engineering, Seoul National University, Seoul, Korea.

L. J. Henschen is with the Department of Electrical Engineering and Computer Sciences, Northwestern University, Evanston, IL 60201.

J. Han is with the School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6 Canada.

IEEE Log Number 9105290.

Among the numerous research issues in deductive databases, our study is focused on classifying and compiling recursive formulas. Classifying recursive formulas has been known to be a hard problem. Therefore, researchers have considered certain specific patterns of recursive formulas that are easily recognizable and compilable [6], [9], [16]. For example, most recursive formulas considered are chained rules [2], [5], [7]. In this paper, we will use a graph model which is useful in classifying and compiling linear recursive formulas in deductive databases. The graph model consists of two kinds of graphs: *I-graph* and *Resolution Graph*. We can extract essential properties of a recursive formula from its I-graph and can easily figure out the compiled formula and the query evaluation plan of the recursive formula from its resolution graph.

We demonstrate that based on the graph model all the linear recursive formulas can be classified into several classes and the formulas in each class share some common characteristics in compilation and query processing. The compiled formulas and the corresponding query evaluation plans can be derived based on the study of the compilation of each class. Our goal is to show a general and uniform query evaluation planning mechanism for each of several important classes of recursive formulas which can map an arbitrary query of that class to a compiled formula. This will illustrate the power and utility of the graph model.

In Section II, after providing a brief introduction to deductive databases, we define the target class of recursive formulas that we deal with in this paper. In Section III, we introduce a graph model that can be used to classify recursive formulas. The I-graph will be used to express and to classify recursive formulas. We also introduce the resolution graph that shows the recursive formula after k expansions. In Section IV, we introduce new terms used in our study and also give an overview of the classification of recursive formulas from a syntactic approach based on the I-graph. In Section V, recursive formulas with one-directional single cycles will be discussed. Most recursive formulas discussed in the literature belong to this class. Recursive formulas having multidirectional single cycles will be dealt with in Section VI. Recursive formulas with acyclic paths (no nontrivial cycles) will be considered in Section VII. Recursive formulas with multiple cycles will be discussed in Section IX. The recursive formulas with one connected component in their I-graphs are studied in Section IV to Section VIII. Recursive formulas having multiple components in their I-graphs will be discussed in Section IX. Section X is allocated for discussing several relevant issues of our study. Section XI concludes the paper.

II. Overture: Deductive Databases

A. First-Order Databases

A first-order database is a deductive database consisting of first-order clauses. We follow the lead of [12]–[14] for preliminaries concerning the terminology and notation of first-order deductive databases. The following symbols will be used in the rest of this paper: Variables: $x, y, z, u, v, w, x_1, y_1, z_1$, ..., Constants : $a, b, c, d, john$, ..., Predicates : A, B, C, D, E, P, Q, R, ... The following example illustrates major concepts of deductive databases.

Example 2.1: Let us discuss an example from [1]. Consider the following first order database regarding ancestor and parent relationships.

1) $Ancestor(x, y) : -Parent(x, z) \wedge Ancestor(z, y)$
2) $Ancestor(x, y) : -Parent(x, y)$
3) $Parent(b, a)$
4) $Parent(c, a)$
5) $Parent(d, b)$
6) $Parent(e, b)$.

In this database, we have a set of predicate or relation names $\{Parent, Ancestor\}$, and a set of constants $\{a, b, c, d, e\}$ and a set of variables $\{x, y, z\}$. The database consists of a set of rules $\{1, 2\}$ and a set of facts $\{3, 4, 5, 6\}$.

Let us associate a meaning to the database. We first associate each constant an object from the real world: thus, to a "a" we associate the individual whose name is "a." Then, we can interpret intuitively each fact and each rule. For instance, we interpret "$Parent(b, a)$" by saying that the predicate $Parent$ is true for the pair (b, a), and we interpret the rule $Ancestor(x, y) : -Parent(x, z) \wedge Ancestor(z, y)$ by saying that if there are three objects x, y and z such that $Parent(x, z)$ is true and $Ancestor(z, y)$ is true then $Ancestor(x, y)$ is true. This leads to the interpretation that associates with each predicate a set of tuples. For instance, with the predicate $Ancestor$ we associate the interpretation $\{(b, a), (c, a), (d, b), (e, b), (d, a), (e, a)\}$. The problem is to answer queries given to the logical database. For instance, for a query of the form $? - Ancestor(?, a)$, how do we find the reply and the Ancester set $\{b, c, d, e\}$?. Q.E.D.

B. The Target Class of Recursive Formulas

We say that a formula is *recursive* if the formula is of the form $P(x) : - \cdots, P(x'), \cdots$ [1] where x and x' are a vector of variables. Among recursive formulas, we restrict our attention to *linear recursive formulas* where the recursive predicate appears only once in the antecedent. Many database researchers believe that most of the recursive formulas in the real world have the recursive predicate appearing only once in the antecedent. We enforce some assumptions on linear recursive formulas.

Rigid Assumptions: Rigid assumptions mean that these assumptions will not be relaxed in this study.

- We deal with only recursive formulas which are *range restricted.* We assume that there are no *arithmetic predicates*, *function symbols* and *negation symbols* in the recursive formulas.
- Recursive formulas with repeated variable are excluded because some results without repeated head variables do not hold in the presence of such variables. Furthermore, many proofs of undecidability use such repeated variables, and cannot be carried through without them [19]. Another side of the coin is that repeated variable in the consequence force us to remember equality. However, the EQ predicate does not come for free. More dicussion on the EQ predicate is provided in Section X.

Many database researchers have been using a set of assumptions on linear recursive formulas which is similar to the rigid assumptions of ours [1], [4], [9]–[11].

Soft Assumptions: We shall relax these soft assumptions later.

- We assume that there are no *constant symbols* in the recursive formula. This restriction "constant-free" is for generality. The variables in the statement can be easily replaced by constants at query evaluation time or from the definition of rules.
- We assume there is only one recursive formula for a given recursive predicate and one or more nonrecursive statements (exit rules) for the given recursive predicate to make our discussion simple. If there are n exit rules and their bodies are $E_1, E_2, \cdots, E_n$, then the exit rule E for the recursive predicate will be $E : -E_1 \cup E_2 \cup, \cdots, \cup E_n$.

There are basically two categories for processing recursive queries: the interpretation approach [14] and the compilation approach [8], [20]. They are well described in [23].

Our goal is to develop compilation techniques for recursive formulas in general. The basic concepts of classification of recursive formulas can be applied directly to the compilation techniques. The graph model used in our research gives us a clear view of recursive rules and shows the connectivity between variables. In compilation techniques, the connectivity of variables is the critical part to reduce query execution time. If variables are connected and one of them is determined (the value of the variable is known at query evaluation time), we can apply selection and join operations to derive values of the remaining variables.

C. The General Compiled Formula

For some typical linear recursive rules, their compiled formulas have been well known from the existing literature such as [6] and [8]. For example, consider the exit rule (r.2.1) and the linear recursive rule (r.2.2).

$$P(x, y) : -A(x, y) \qquad \text{(r.2.1)}$$
$$P(x, y) : -B(x, z) \wedge P(z, w) \wedge C(w, y) \qquad \text{(r.2.2)}$$

where A, B, and C are base relations. Consider a recursive query $? - P(a, y)$ where a is a constant and y is the variable to be retrieved. Using stepwise recursive calls on the recursive rules, we obtain a sequence of expressions

$$P(x,y) :- B(x,z) \wedge B(z,z_1) \wedge P(z_1,w_1) \wedge C(w_1,w) \wedge C(w,y)$$

$$\cdots$$

$$P(x,y) :- B(x,z) \wedge \cdots \wedge B(z_{k-1},z_k) \wedge P(z_k,w_k) \wedge C(w_k,w_{k-1}) \wedge \cdots \wedge C(w,y).$$

Using the exit rule on the above sequence, the solutions are obtained by processing the following sequence of formulas and unioning the results.

$$A(a,y)$$

$$B(a,z) \wedge A(z,w) \wedge C(w,y)$$

$$B(a,z) \wedge B(z,z_1) \wedge A(z_1,w_1) \wedge C(w_1,w) \wedge C(w,y)$$

$$\cdots$$

$$B(a,z) \wedge \cdots \wedge B(z_{k-1},z_k) \wedge A(z_k,w_k) \wedge \cdots \wedge C(w_k,w_{k-1}) \wedge \cdots \wedge C(w,y).$$

The expansion sequence terminates when no new values for z_k are found in the database. The sequence can be represented with a compiled formula $\sigma_a B^k \bowtie A \bowtie C^k$ where $\sigma_a B$ means the selection of a on the corresponding attribute of relation B, $\bowtie$ means the join operation on the corresponding attributes, and B^k means the join formed by k B relations. Join attributes are omitted in the formula for simplicity, and the range of index k is from 0 to n where n is the number of iterations up to the termination point, i.e., up to the first k where $\sigma_a B^k$ is empty as long as the given relation B is acyclic. In what follows, when no ambiguity results, we can simply write $\sigma B^k A C^k$. In this paper, we shall use this general compiled formula notation to express the flow of information and query evaluation plans.

III. The Graph Model

Several graph models have been proposed for understanding behaviors and various properties of recursive formulas: I-graph [9], A/V graph [16], Augmented A/V graph [18], Substitution graph [11], Rule/Goal graph [23], and etc.

Our graph model consists of two kinds of graphs: *I-graph* and *Resolution graph.* In this section, we give definitions of I-graph which will be used as a tool to represent and analyze recursive formulas. The I-graph was invented originally by Ioannidis [9], but he used this graph model only to derive the condition of boundness (to be defined shortly) for recursive formulas. For studying the generic properties of recursive formulas we shall use the I-graph in a broader way than Ioannidis did. We also introduce the *resolution graph* that is used to derive compiled formulas and query evaluation plans of recursive formulas.

A. The I-graph

Definition 3.1: Suppose we are given a recursive statement F. We will associate a labeled, weighted, hybrid graph $G_1 = (V, E_u, E_d, W, \mathcal{L})$ to F. Since the graph construction was originally introduced by Ioannidis, from now we call this variable connection graph, *I-graph.* Following is the definition of I-graph.

a) V is a finite, nonempty set of nodes such that a node is defined and labeled by x if x is a variable in F.

b) E_u is a finite, nonempty set of undirected edges such that an undirected edge, referred to as an undirected self-loop, is defined, denoted by (x–x), and labeled by Q if x is a node in V corresponding to the variable of a unary, nonrecursive predicate Q; and an undirected edge is defined, denoted by (x–y), and labeled by Q for each pair of nodes x, y in V corresponding to a pair of distinct variables x, y of an n-ary nonrecursive predicate Q for $n > 1$.

c) E_d is a finte, nonempty set of directed edges such that for each pair of (nonnecessarily distinct) nodes x, y in V a directed edge is defined, denoted by ($x \rightarrow y$), and labeled by P if x, y are nodes in V where x and y correspond, respectively to the ith variable of the the recursive predicate P in the head and in the body. When $x = y$, the directed edge is referred to as a directed self-loop. For each directed edge ($x \rightarrow y$) that is not a directed self-loop, the implicit reverse directed edge ($y \rightarrow x$) is defined.

Comment: $E_u \cap E_d = \phi$ and $E_u \cup E_d = E$.

d) $W : E \longrightarrow \{0, 1\}$. W is a weight function that associates a weight to each edge. If x occurs in the consequent and y in the antecedent, then the directed edge ($x \longrightarrow y$) has weight 1 and its reverse edge ($y \longrightarrow x$) has weight -1. Each undirected edge (x–y) has weight 0.

e) $\mathcal{L} : E \longrightarrow L$ where $\mathcal{L}$ is a label function that associates a label to each edge. An undirected edge has a label Q if two nodes of the undirected edge are in the nonrecursive predicate Q. Each directed edge has a label P where P is the recursive predicate.

Definition 3.2: The *weight* of a path (cycle) in the graph is defined as the algebraic sum of the weights of the edges along the path (cycle). Regarding undirected edges, they can be traversed in either direction. Traversing a directed edge in the opposite direction of the arrow is the same as traversing the implicit reverse edge and contributes -1 to the weight.

Remark: For a variable in a unary, nonrecursive predicate, there is an undirected self-loop (x, x).

Example 3.1: Consider the following two recursive formulas.

$$P(x,y) :- A(x,z) \wedge P(z,y) \qquad \text{(r.3.1)}$$

$$P(x,y,z) :- A(x,y) \wedge P(u,z,v) \wedge B(u,v). \qquad \text{(r.3.2)}$$

The corresponding I-graphs of the rules (r.3.1) and (r.3.2) are in Fig. 1(a) and (b), respectively. We do not write down the label P for directed edges because there is only one recursive predicate for each formula and all directed edges have the same label P. Q.E.D.

B. The Resolution Graph

We introduce the *resolution graph* to represent formulas obtained by repeated application of recursive formulas. These

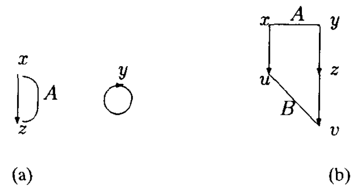


Fig. 1. I-graphs of the rules (r.3.1) and (r.3.2).

graphs contain extra information that is useful in analyzing recursive query plans.

Definition 3.3: For a given recursive formula F, the recursive formula made by expanding the recursive predicate k times is called the *kth expansion* of F. The original recursive formula is *the first expansion* of F.

Remark: We can consider the exit rule as the 0th expansion of F.

Clearly, the kth expansion ($k \geq 2$) is formed by resolving the ($k - 1$)st expansion with the original rule. This involves renumbering the original rule and unifying its consequent with the recursive predicate in the antecedent of the ($k - 1$)st expansion. In order to easily draw graphs for the expanded formulas, we choose to unify by substituting for the variables in the renumbered rule, leaving the ($k - 1$)st expansion unchanged.

Definition 3.4: The formula obtained by renumbering and unifying the original recursive formula which is used for the kth expansion of F is called the *kth unified formula* of F. The original formula is the *first unified formula* of F.

Definition 3.5: The I-graph drawn from the kth unified formula is called the *kth I-graph.* The I-graph from the original formula is the *first I-graph* or simply, I-graph.

Definition 3.6: Let us consider a graph for the kth expansion of a formula F. The *kth resolution graph*, $G_k = (V, E_u, E_d, W, \mathcal{L})$ of F is defined recursively in the following manner.

1) The I-graph of F is the first resolution graph.
2) The kth resolution graph, $G_k (k \geq 2)$ of F is obtained from the ($k - 1$)st resolution graph, G_{k-1}, by the following process.
   2.1) Form a kth I-graph from the kth unified formula.
   2.2) Append the kth I-graph to the ($k - 1$)st resolution graph using common variables.

The above definition on the resolution graph is actually only for recursive formulas without repeated variables. The kth resolution graph retains all the arrows from the ($k - 1$)st I-graph. Further, the kth resolution graph is formed directly from the ($k - 1$)st resolution graph without the need to actually form a resolvent. These retained directed edges give a better picture of the derivation.

Example 3.2: Consider the following recursive formula.

$$P(x, y) : -A(x, z) \wedge P(z, u) \wedge B(u, y). \qquad \text{(r.3.3)}$$

The I-graph is shown in Fig. 2(a). By renaming the variables in the rule (r.3.3), we have

$$P(z, u) : -A(z, z_1) \wedge P(z_1, u_1) \wedge B(u_1, u). \qquad \text{(r.3.4)}$$

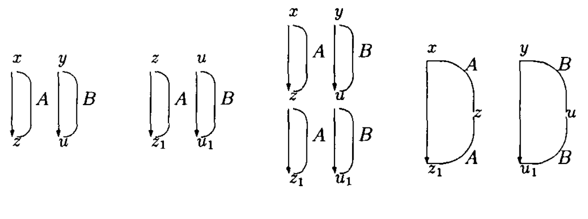


Fig. 2. I-graphs and resolution graphs for (r.3.3).

By unification, we get the second expansion of the rule (r.3.3). [This is shown in Fig. 2(c)]:

$$P(x, y) : -A(x, z) \wedge A(z, z_1) \wedge P(z_1, u_1) \wedge B(u_1, u) \wedge B(u, y). \qquad \text{(r.3.5)}$$

The second I-graph of the rule (r.3.3) is in Fig. 2(b). The second resolution graph G_2 in Fig. 2(c) can be drawn by appending Fig. 2(b) to 2(a). In Fig. 2(c), the weight from x to z_1 is two. That means, in the second expansion of the rule (r.3.3), x appears under the recursive predicate P in the consequent and z_1 appears in the corresponding position of the recursive predicate in the antecedent. We have Fig. 2(d) by considering the rule (r.3.5) as a formula by itself as opposed to a second resolution graph. Q.E.D.

Example 3.3: Consider another linear recursive rule.

$$P(x, y, z) : -A(x, y) \wedge P(u, z, v) \wedge B(u, v). \qquad \text{(r.3.6)}$$

The second expansion of the rule (r.3.6) is

$$P(x, y, z) : - A(x, y) \wedge A(u, z) \wedge P(u_1, v, v_1) \wedge B(u_1, v_1) \wedge B(u, v). \qquad \text{(r.3.7)}$$

Fig. 3(a) and (b) are the first and second resolution graphs. The third resolution graph is easily drawn by appending an isomorphic renumbered copy of the graph in Fig. 3(a) to the variables u_1, v, v_1 in the second resolution graph. Q.E.D.

This graph model is a powerful tool for explaining and formalizing recursive formulas. Based on the resolution graph, we can easily detect the flow of information which is essential in finding out query evaluation plans. Undirected edges are used to send information from one variable to another. Therefore, we can easily derive compiled formulas and query evaluation plans from the resolution graph.

IV. Classification of Recursive Formulas

In this section, we define new terms we use to classify recursive formulas. A classification taxonomy of recursive formulas is introduced.

A. New Terms

Definition 4.1: The *dimension* (D) of a recursive formula is the number of variables in the recursive predicate. If there are n variables in the recursive predicate, we call the formula an n-D recursive statement.

Remark: There are 2^n different query forms possible on an n-D recursive formula by assigning a constant or a variable in each position.

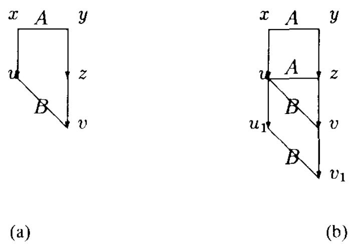


Fig. 3. Resolution graphs for (r.3.7).

Definition 4.2: A variable in the recursive formula F (after kth expansion) is a *determined variable* for a given query if the value of the variable is given in the query or derivable from a query constant by selection and join operations over only the nonrecursive predicates in the (kth resolution) graph. If x is a determined variable and $L(\ldots x..y..)$ is a nonrecursive predicate, then y is also a determined variable [8].

Definition 4.3: We can define *connected* by the following:

1) Two variables are connected if there is an undirected edge between the two variables.
2) If variables x and y are connected, and variables y and z are connected, then variables x and z are connected.

Definition 4.4: A *nontrivial component* is a component of the I-graph with at least one directed edge. Otherwise, the component is a *trivial component.*

Definition 4.5: A cycle is *trivial* if it has no directed edges. Otherwise, the cycle is *nontrivial.*

Example 4.1: For example, the component in Fig. 4(a) is a nontrivial component (cycle) and the component in Fig. 4(b) is a trivial component (cycle). Q.E.D.

Remark: If there is a trivial component, this component will be used only for existence checking, and we will not consider the trivial component in our discussion.

Definition 4.6: A nontrivial cycle is *pure* if it is not connected to any other nontrivial cycles nor to any other directed edges. Otherwise, the cycle is *dependent.*

Example 4.2: For example, the cycles in Fig. 4(a) and Fig. 5(a) are pure cycles and the cycle in Fig. 5(b) is a dependent cycle. It is assumed in Fig. 5(a) that A and B can be reduced to C where $C(x, y) = A(x, y) \cap B(x, y)$. More detailed simplification of nonrecursive components can be found in [24] Q.E.D.

Definition 4.7: A pure cycle is *one-directional* if all the directed edges along the cycle have the same direction. Otherwise, the pure cycle is *multidirectional.*

Example 4.3: For example, the cycle in Fig. 6(a) is a pure one-directional cycle and the cycle in Fig. 6(b) is a pure multidirectional cycle. Q.E.D.

Definition 4.8: A pure one-directional cycle is *rotational* if there is at least one undirected edge as a part of the nontrivial cycle. Otherwise, the pure one-directional cycle is *permutational.*

Example 4.4: For example, the cycle in Fig. 6(a) is a pure one-directional rotational cycle and cycles in Fig. 7 are pure one-directional permutational cycles. Q.E.D.

Remark: If there is no undirected edge as a part of the

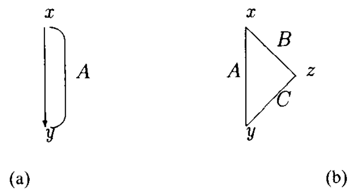


Fig. 4. Trivial and nontrivial cycles.

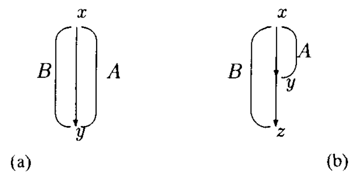


Fig. 5. Pure and dependent cycles.

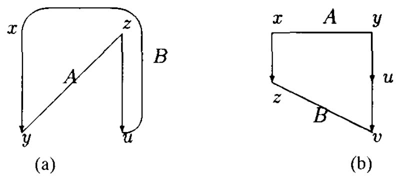


Fig. 6. One-directional versus multidirectional cycles.

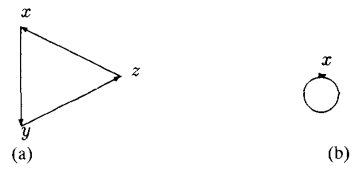


Fig. 7. Nonunit and unit cycles.

cycle, all the variables of the recursive predicate in the antecedent are from the variables of the recursive predicate in the consequent, and we call this pattern *permutational* because the recursive predicate in the antecedent is made by simply changing the order of the variables of the recursive predicate in the consequent.

Definition 4.9: A pure one-directional cycle is a *unit cycle* if the weight of the cycle is 1. Otherwise, the cycle is a *nonunit* cycle.

Example 4.5: For example, the pure one-directional cycles in Fig. 4(a), Fig. 5(a), and Fig. 7(b) are unit cycles and the pure one-directional cycles in Fig. 6(a) and Fig. 7(a) are nonunit cycles. Q.E.D.

B. A Classification Taxonomy

If we look at the I-graphs from the syntactic view point using the definitions above, we can have the following cases.

(1) Acyclic Paths
(2) Pure Cycles
  (2.1) Pure One-Directional Cycles

(2.1.1) Pure One-Directional Unit Rotational Cycles
(2.1.2) Pure One-Directional Unit Permutational Cycles
(2.1.3) Pure One-Directional Nonunit Rotational Cycles
(2.1.4) Pure One-Directional Nonunit Permutational Cycles
(2.2) Pure Multidirectional Cycles
(3) Dependent Cycles
(4) Heterogeneous Components

Category (1) is for recursive formulas whose I-graphs have one or more components having no nontrivial cycle (i.e., an acyclic path). Category (2) is for recursive formulas whose I-graphs have one or more components having a pure cycle. Category (3) is for recursive formulas whose I-graphs have one or more components having a dependent cycle. If an I-graph has components which are composed of disjoint combinations of different categories, we classify the I-graph into (4). At the first glance, we can say that the taxonomy is exhaustive. In the later section, we will show the completeness of our taxonomy.

We can classify all possible linear recursive formulas using the taxonomy above. There are different properties (semantics) for each of the classes, i.e., compiled formulas and query evaluation plans are quite different. Properties of each class will be fully discussed in the following sections.

Even though the taxonomy is syntactic, the order of our presentation is based on the semantic significance of each class. Since the cases in (2.1) "Pure One-Directional Cycles" are the most important subset of real-world linear recursive formulas, we start with (2.1) in Section V. Then we will discuss (2.2) "Pure Multi-Directional Cycles" in Section VI. After that, we will discuss 1 "Acyclic Paths" in Section VII. Then we will cover 3 "Dependent Cycles" and (4) "Heterogeneous Components" in Sections VIII and IX, respectively. Some similar limited classification can be found in [17] and [18].

V. Pure One-Directional Cycles (2.1)

Many recursive formulas in real world applications are classified as pure one-directional cycles. In this section we investigate the properties of linear recursive formulas whose I-graphs form pure one-directional cycles.

A. Pure One-Directional Unit Cycles (2.1.1 and 2.1.2)

There are two different kinds of pure one-directional unit cycles; one is the class of *pure one-directional unit rotational* cycles [class (2.1.1)] and the other one is the class of *pure one-directional unit permutational* cycles [class (2.1.2)]. A pure one-directional unit rotational cycle is a unit cycle with at least one undirected edge and by recursive expansions, new variables will be generated for the recursive predicate. A pure one-directional unit permutational cycle is a directed self loop and by recursive expansions, no new variables will be generated for the recursive predicate.

Definition 5.1: A recursive formula is *strongly stable* if there are only disjoint pure one-directional unit cycles in the corresponding I-graph.

Theorem 5.1: A recursive formula is strongly stable if and only if the determined variables of the recursive predicate in the consequent and in the antecedent occur in the same positions for any query.

Proof: (Only-If part): A unit cycle can be a directed self loop (permutational) or a cycle with undirected edges (rotational). Recall, such a cycle has only one directed edge, and, since it is a cycle, must therefore have at least one undirected edge or be a self loop. In both cases, if a variable in the consequent is determined, then the variable in the same position in the antecedent will be determined, and no variables in other positions in the antecedent will be determined because all cycles are disjoint. By induction on number of expansions, we can easily see that stability will be preserved for arbitrary numbers of expansions.

(If part): Suppose the graph is not stable because of a one-directional cycle of length two, say $P(x, y) : -A(x, z) \wedge P(y, z)$. A query in which only x is determined gives a determined variable z in a different position in the antecedent. A similar nonsatisfactory query form can be found if the cycle is not one-directional or fails the stability condition in any other way. Thus, if the determined variables of the recursive predicate in the consequent and in the antecedent occur in the same position on arbitrary queries, then there are n disjoint connections for n-D recursive formula and each connection is made between variables in the same position of the recursive predicate in the consequent and in the antecedent. Therefore, there are n disjoint unit cycles. Q.E.D.

Thus we have equivalent syntactic and semantic *characterizations* for strongly stable formulas.

Remark: Note that this definition of strongly stable is stronger than that given in [8]. Here, the condition on determined variables holds for *all* query forms. From now on, we call a strongly stable formula simply a "stable formula."

An n-D stable recursive formula can be viewed as in Fig. 8. There is one relation for each disjoint cycle and they are named $A_1, A_2, \cdots, A_n$. (Any A_i will be an empty relation if the corresponding cycle is permutational, otherwise the cycle is rotational.) Each relation A_i $(1 \leq i \leq n)$ is connected only to the exit relation E. If a query is given, selection operations can be applied to all the A_i relations which have determined variables. If the corresponding A_i is empty, then selection is applied to the exit relation. The A_i that are so determined will then be written on the left of E and the remaining ones on the right in the evaluation plans described below.

Many recursive formulas in real systems are 2-D stable formulas. The compiled formula of a 2-D stable formula can be expressed as

$$\bigcup_{k=0}^{\infty} A'^k - E - B'^k. \qquad \text{(r.5.1)}$$

A' and B' are the combined relations from the two disjoint components respectively (any one of them can be an empty relation). The symbol "-" is used for the connectivity between

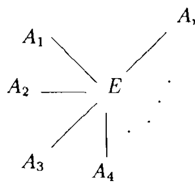


Fig. 8. An n-D stable recursive formula.

literals and can be omitted in the expression. The symbol "$\bigcup$" is used for the union of all possible answers for different k's.

Example 5.2: Consider the following recursive rule.

$$P(x,y) : -A(x,z) \wedge P(z,u) \wedge B(u,y). \qquad \text{(r.5.2)}$$

There are two disjoint pure one-directional unit cycles in the corresponding I-graph, and the formula (r.5.2) is a 2-D stable formula. The corresponding resolution graphs are shown in Fig. 2 in Section III. If a query $? - P(a,Y)$ is given, we can easily see the flow of information based on the resolution graphs. The variable x is known from the query, and all the variables in the predicates A are determined (because all the variables of A are connected to the variable x). We can apply the selection operation (the symbol "σ") on the relation A followed by join operations. The evaluation plan for this query is $\bigcup_{k=0}^{\infty} \sigma A^k \bowtie E \bowtie B^k$. The notation $\sigma A^k \bowtie E \bowtie B^k$ means that the evaluation will be done from left to right. If a query $? - P(X,b)$ is given, all the variables in B are determined and the evaluation plan for the given query is $\bigcup_{k=0}^{\infty} \sigma B^k \bowtie E \bowtie A^k$. We note that, independent of the query, each kth expression will have A^k and B^k plus E. Thus, we may write $\bigcup_{k=0}^{\infty} B^k E A^k$ (or $\bigcup_{k=0}^{\infty} A^k E B^k$) without any selection or any implied order of evaluation as the general compiled formula. Then, given a query, the actual evaluation plan can be easily derived. Q.E.D.

Definition 5.2: A *transitive closure rule* is a linear recursive rule whose I-graph contains exactly one pure one-directional unit rotational cycle and zero or more unit permutational cycles.

If we consider the transitive closure formula $P(x,y) : -A(x,z) \wedge P(z,y)$ by removing the relation B from the formula (r.5.2), the compiled formula is: A^*E and the query evaluation plan for the query $? - P(x,b)$ is $\sigma E A^*$.

Many recursive formulas in real systems are 2-D stable formulas. Researchers have considered some of them as chained rules. The join operation is considered as a chain that connects two neighboring relations, and if there are shared variables in neighboring relations, join operations can be made at query processing time.

Example 5.2: Consider the following recursive formula.

$$P(x,y,z) : -A(x,u) \wedge B(y,v) \wedge P(u,v,w) \wedge C(w,z). \qquad \text{(r.5.3)}$$

The corresponding I-graph shows that there are three disjoint pure one-directional unit cycles. Therefore, the rule (r.5.3) is a stable formula. The compiled formula and the evaluation plan for the query $? - P(a,b,z)$ are in Fig. 9. A branch such as $\begin{smallmatrix}\sigma A^k \\ \sigma B^k\end{smallmatrix} >$ means that A^k and B^k are evaluated independently (we use "-" for the join operation because of the difficulty to use the symbol "$\bowtie$"), and the results are then combined with E. We can find evaluation plans for other possible queries (e.g. $? - P(x,b,c)$ and $? - P(x,a,z)$) in a similar way. Q.E.D.

For recursive formulas of this class, the compiled formulas are easily obtained and query evaluation plans for all possible queries are also easily found. As such, we are dealing on the higher level logical form so that the global query plan can be optimized. Once a compiled formula is obtained, actual query processing can be done in many different ways such as *Henschen–Naqvi method*, *Counting method*, *Magic set method*, etc [1].

B. Pure One-Directional Nonunit Cycles (2.1.3 and 2.1.4)

Classes (2.1.3) and (2.1.4) consist of recursive formulas with pure one-directional nonunit cycles, which are *rotational* or *permutational*, respectively. Interestingly, the recursive formulas in pure one-directional nonunit cycles can be transformed into recursive formulas with pure one-directional unit cycles. The idea is that even if a recursive formula is not stable itself, the recursive formula can be transformed to an equivalent set of recursive formulas which does represent stable recursion and the query compilation techniques of stable formulas can still be applied. To that end, we show general methods by which recursive formulas in (2.1.3) and (2.1.4) can be so transformed.

Theorem 5.2: Suppose there is a pure one-directional cycle of weight n in the I-graph for an n-D recursive formula, F. Let F' be the result by expanding F n times. Then

1) F' is a stable formula.
2) F' along with additional exit rules is equivalent to F.

Proof: 1): If two directed edges share a variable, we can assume there is a nonrecursive predicate "EQUAL" between the shared variable (this is only for the theoretical development). If there is a trivial cycle or if there is more than one undirected edge shared by the same variables, we can collapse them into a single undirected edge. Therefore, without loss of generality, we can assume that there is exactly one undirected edge (nonrecursive predicate) between directed edges in the cycle of weight n. Let us call the variables in the consequent $x_1, x_2, \cdots, x_n$ (by the traversal order of the cycle) and the corresponding variables in the antecedent $y_1, y_2, \cdots, y_n$. In the first expansion, x_n is connected to y_1 and to no other y. By the following expansion, new variables $z_1, z_2, z_3, \cdots, z_n$ will be produced and x_n is connected with z_2 and to no other z. By induction on the number of expansions, we can easily find that after n expansions, x_n will be connected to the variable in same position and will not be connected to the variables in any other positions. The second resolution graph is in Fig. 10, and the connection from x_n to z_2 is shown in the figure.

2): From property (1), we find the cyclic behavior of the formula. Generate the first $(n-1)$ expansions of F and replace the recursive predicate in the antecedents by the exit relation and leave the nth expansion of F as a new recursive formula. The new recursive formula with n exit relations is stable and produces the same answers as the original formula. It is in fact, logically equivalent to the original set. Q.E.D.

$$\bigcup_{k=0}^{\infty}\begin{tikzcd}[arrows=-] & B^k \\ A^k \arrow[r] & E \arrow[u] \arrow[d] \\ & C^k \end{tikzcd}, \qquad \bigcup_{k=0}^{\infty}\begin{tikzcd}[arrows=-] \sigma A^k & \\ & E \arrow[ul] \arrow[dl] \arrow[r] & C^k \\ \sigma B^k & \end{tikzcd}$$

Fig. 9. The compiled formula and an evaluation plan for (r.5.3).

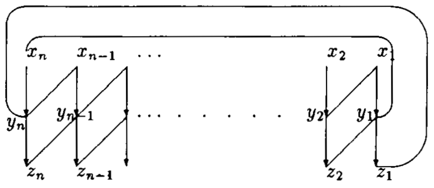


Fig. 10. Resolution graph pattern for recursive formulas with pure one-directional cycles.

We now know that a formula F with a pure cycle of weight n becomes a stable formula after n expansions, and we can consider that the nth resolution graph of F has n disjoint unit cycles. Sample rules are mentioned in the following.

Corollary 5.1: A recursive formula with pure one-directional nonunit rotational cycles can be transformed to a stable recursive formula.

Proof: From the Theorem 5.2. Q.E.D.

Example 5.3: Consider the following rules.

$$P(x,y) : -E(x,y) \qquad \text{(r.5.1)}$$
$$P(x,y) : -A(x,u) \wedge P(v,u) \wedge B(v,y). \qquad \text{(r.5.2)}$$

The I-graph of the recursive formula (r.5.2) shows that the cycle is rotational with weight 2 and the recursive formula is nonstable. The resolution graphs after the first, second, and third expansion are in Fig. 11(a), (b), and (c), respectively. As we can see in the graph, the formula becomes stable after every two expansions. Formulas (r.5.3) and (r.5.4) can be transformed to an equivalent stable formula by unfolding exactly two times. Therefore, the original rule is equivalent to one recursive rule (r.5.3), and two exit rules (r.5.4) and (r.5.5).

$$P(x,y) : -A(x,u) \wedge B(v_1,u) \wedge P(v_1,u_1) \wedge A(v,u_1) \wedge B(v,y) \qquad \text{(r.5.3)}$$
$$P(x,y) : -A(x,u) \wedge E(v,u) \wedge B(v,y) \qquad \text{(r.5.4)}$$
$$P(x,y) : -E(x,y). \qquad \text{(r.5.5)}$$

The I-graphs for rules (r.5.2) and (r.5.3) are in Fig. 11(a) and Fig. 11(b), respectively. The compiled formula of rules (r.5.1) and (r.5.2) is $\bigcup_{k=0}^{\infty}(AB)^k - (E \cup AEB) - (AB)^k$ and the evaluation plan for the query $? - P(x,b)$ is $\bigcup_{k=0}^{\infty} \sigma(BA)^k \bowtie (E \cup BEA) \bowtie (BA)^k$. In the compiled formula, the notation "AB" indicates the connectivity of two relations and does not indicate any particular order. In a query evaluation plan, however, order of the predicates is the actual order to be used in the evaluation process.

We can consider an example where the number of undirected edges is less than the number of directed edges. For example,

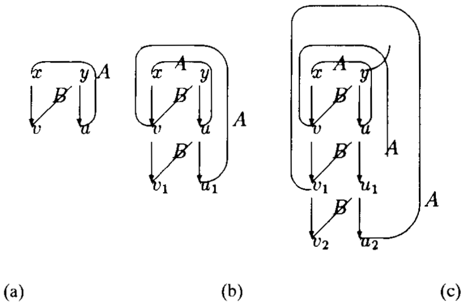


Fig. 11. Resolution graphs for (r.5.2)

the recursive formula $P(x,y) : -A(x,u) \wedge P(y,u)$ has the compiled formula: $\bigcup_{k=0}^{\infty} A^k(E \cup AE)A^k$. Q.E.D.

We need to introduce the notion of *boundness* because, if the I-graph of a recursive rule consists of only pure one-directional permutational cycles, the corresponding recursion is bounded. Some recursive formulas have an upper bound on the number of iterations necessary to form a virtual relation (i.e., to answer a given recursive query), independent of the contents of the base relations [9], [16]. More discussion on bounded recursions will be given in Sections VII and VIII.

A popular example of bounded recursion is *Spouse*, where $Spouse(x,y) : -Spouse(y,x)$. The I-graph [in Fig. 12(a)] has one permutational cycle of weight 2, and it is bounded since no new tuples can be generated after one expansion.

Corollary 5.2: A recursive formula with a pure one-directional nonunit permutational cycle can be transformed to a stable formula.

Proof: From the Theorem 5.2. Q.E.D.

Example 5.4: Consider the following recursive formula.

$$P(x,y,z) : -P(y,z,x).$$

The corresponding I-graph in Fig. 12(b) shows that there is a cycle of weight three. The recursive formula can be transformed into a stable formula. But there is no nonrecursive relation involved in the expansions, and after two expansions, the formula cannot produce any new values (or tuples). As we mentioned above, we call such recursive formulas bounded. Bounded formulas will not produce any new tuples (values) after certain expansions regardless of the contents of the database. The above formula has no new variables in the antecedent; all the variables are from the consequent. This is called a "permutational pattern" to distinguish it from rotational formulas. Q.E.D.

VI. Pure Multidirectional Cycles (2.2)

In this section, we will discuss recursive formulas that have pure multidirectional cycles. As we discussed before, to be transformed into a stable formula, a recursive formula should have a pure one-directional unit cycle after some finite expansions. As we can see by the following theorem, pure multidirectional cycles cannot be transformed into pure one-directional unit cycles. Therefore, recursive formulas with pure multidirectional cycles cannot be transformed into stable formulas.

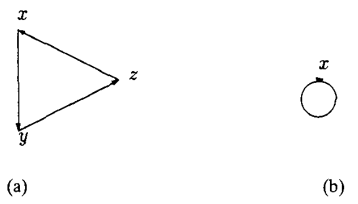


Fig. 12. I-graphs with permutational cycles.

Theorem 6.1: A recursive formula with pure multidirectional cycle cannot be transformed to a strongly stable formula.

Proof: A pure multidirectional cycle has at least one (possibly compressed) undirected edge whose two nodes each are used as the tail of directed edges. By the expansion of the resolution graph, the two nodes can never be split, and the resolution graph cannot be expressed as disjoint unit cycles (refer to the Theorem 6.2). Therefore, the formula cannot be transformed to a stable formula. Q.E.D.

Corollary 6.1: A recursive formula with pure cycles can be transformed to a strongly stable formula (or is stable) if and only if it is one-directional.

Proof: From the Theorem 6.1. Q.E.D.

Recursive formulas with pure multidirectional cycles can be semantically divided into two subclasses: Pure Multidirectional Bounded Cycle and Pure Multidirectional Unbounded Cycle.

$P(x, y) : -A(x, y) \wedge E(u, v) \wedge B(u, v).$ Q.E.D.

A. Pure Multidirectional Bounded Cycle

Definition 6.1: A pure cycle is called a *bounded cycle* if the weight of the cycle is 0.

Definition 6.2: The *rank* of a recursive formula is defined to be the smallest i such that the $(i + 1)$st expansion and all succeeding expansions do not produce any tuple not found in the first i expansions.

Definition 6.3: A recursive formula is called *bounded* if and only if there exists a finite upper bound on its rank independent of the contents of the relations involved in the formula.

Ioannidis's Boundness Theorem [9]: Let F be a recursive formula with no permutational patterns. Then F is bounded if and only if the corresponding I-graph contains no cycle of nonzero weight. In that case a tight upper bound on the rank of the recursive formula is given by the maximum weight of any path in the I-graph.

Corollary 6.2: A bounded cycle is bounded.

Proof: From the Ioannidis's boundness theorem. Q.E.D.

A pure multidirectional cycle in the I-graph of a recursive rule indicates that we will encounter directed edges in different directions when traversing a cycle in one way. Since we assume that there are no repeated variables in the recursive predicate, a pure multidirectional cycle must contain at least one undirected edge (otherwise, there must be a vertex being the tails (heads) of two directed edges, which violate the assumption). This implies that a pure multidirectional cycle has at least one undirected edge in it.

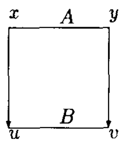


Fig. 13. I-graph for (r.6.1).

Example 6.1: Consider the following recursive formula.

$$P(x, y) : -A(x, y) \wedge P(u, v) \wedge B(u, v). \qquad \text{(r.6.1)}$$

The corresponding I-graph is shown in Fig. 13. There are no cycles of nonzero weight in the I-graph of the rule (r.6.1) including those formed through the negative directed edges, which according to our convention are not drawn (going along a negative edge can be thought of as going along a positive edge in the opposite direction and inverting the weight). Therefore, the formula (r.6.1) is bounded and the least upper bound is 1 (because the maximum weight of any path in the I-graph is 1). Recursion of bounded formulas is sometimes called "pseudo recursion" because the answers to the bounded formulas are obtained without actual recursion. If a recursive formula is bounded, there is an equivalent finite set of nonrecursive formulas, e.g., the formula (r.6.1) has the least upper bound 1, and will be expressed as nonrecursive formula(s) by replacing relation P in the antecedent by the exit relation E: $P(x, y) : -A(x, y) \wedge E(u, v) \wedge B(u, v)$.Q.E.D.

Another general characteristic of the graphs of the bounded statements is that the last expansion that is significant (regarding the production of new tuples) is the first one with the maximum weight of any path in its graph being 1 [9].

Theorem 6.2: A formula constructed by a disjoint combination of bounded cycles is bounded.

Proof: All the disjoint components will be expanded independently, and all the components are bounded. Therefore, the formula is bounded. Q.E.D.

Bounded formulas have been considered by many researchers for optimization because after certain expansions we do not need to generate further expansions of the formula nor do further query processing. The general solution providing the upper bound is fully mentioned by [9].

B. Pure Multidirectional Unbounded Cycle

Definition 6.4: A pure multidirectional cycle of nonzero weight is called a pure multidirectional unbounded cycle.

Query evaluation plans for pure multidirectional unbounded cycles are more complicated than the previous cases. In fact, a general compiled formula (such as $A^k E B^k$ for strongly stable recursive formulas) is not known for recursive formulas with pure multidirectional unbounded cycles at this time. It is not likely to find such general compiled formulas. But if we use the resolution graph, we can easily derive compiled formulas (or query evaluation plans) for individual cases.

Example 6.2: Consider the following recursive formula.

$$P(x, y, z) : -A(x, y) \wedge B(u, v) \wedge P(u, z, v). \qquad \text{(r.6.2)}$$

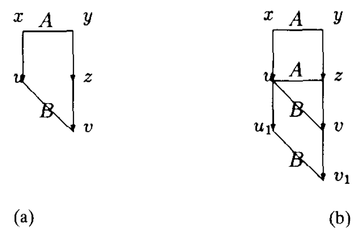


Fig. 14. I-graph for (r.6.2).

The first and second resolution graphs are in Fig. 14. Let us consider a query of the form $? - P(d, v, v)$, where v stands for variables and d for a data value, either from the query or from recursive predicates after some expansions. The evaluation process will be the following:

The first expansion [Fig. 14(a)]: The value of x is given and we can apply the selection operation to the relation A and derive values of y following undirected edges. There is no more selection or join operation possible. If there is no information available, we will select the exit relation, E, (this strategy is the conventional technique) and derive all the tuples of E. Then we can apply the join operation with the relation B to find values of z. The answer will be the Cartesian product (symbol $\times$ will be used) of values of y and z. This evaluation step can be expressed as $(\sigma A) \times (E \bowtie B)$.

The second expansion [Fig. 14(b)]: The value of x is given and we can derive values of y following undirected edges as in the first expansion. There is no more selection and join operation possible. We will derive all the tuples of exit relation E. Then we can apply the join operation with B to find common tuples (for variables u_1 and v_1), and apply the join with B and A successively to find values of z. Evaluation steps will be $(\sigma A) \times [(E \bowtie B)BA]$.

The query evaluation plan for a query $? - P(d, v, v)$ can be expressed as $\sigma E, (\sigma A) \times (\bigcup_{k=0}^{\infty}[(E \bowtie B)(BA)^k])$. For a query of the form $? - P(v, v, d)$, we have the following evaluation plan from the resolution graphs: $\sigma E, (\exists \bigcup_{k=0}^{\infty}[(AB)^k(E \bowtie B)])A$. The symbol "$\exists$" is used for the existence checking for the immediately following expression. This means if there is any tuple (not empty) that satisfies the expression in the $(\exists \cdots)$, then all the tuples in the relation A will be answers. Q.E.D.

VII. Acyclic Paths (1)

In this section, we will consider components with no nontrivial cycles (acyclic paths).

Theorem 7.1: A recursive formula with no nontrivial cycle cannot be transformed to a strongly stable formula.

Proof: 1): Suppose that there is only one directed edge. The head and tail of the directed edge are not connected to each other by any undirected edge(s). By induction on the number of expansions, the head and tail will never be connected.

2): If there is more than one directed edge, there are two possibilities, a) directed edges are one directional, b) directed edges are multidirectional. In case a), there is a leftmost (or rightmost) node used as a tail of a directed edge. By induction on the number of expansions, the node will never be connected to any other nodes, and the formula cannot be transformed to a stable one. In case b), we can prove as in the previous theorem. Q.E.D.

Corollary 7.1: A component with no nontrivial cycle is bounded.

Proof: There is no cycle of nonzero weight in this component. From Ioannidis's theorem, the component is bounded. Q.E.D.

Example 7.1: Consider the following formula.

$$P(x, y) : -B(y) \wedge C(x, y_1) \wedge P(x_1, y_1). \qquad \text{(r.7.1)}$$

There are no nontrivial cycles in the I-graph (Fig. 15) for the formula (r.7.1). If a query $? - P(x, y)$ is given, we can derive all the tuples from $E(x, y)$ and from $B(y) \wedge C(x, y_1) \wedge E(x_1, y_1)$ and, finally from $B(y) \wedge C(x, y_1) \wedge B(y_1) \wedge C(x_1, y_2) \wedge E(x_2, y_2)$.

Further expansions will not produce any new tuples and the upper bound is 2. Q.E.D.

VIII. Dependent Cycles (3)

Although so far there are no complete general techniques for finding compiled formulas developed for this class, we will show by examples that the I-graph and the resolution graph can be very useful in obtaining query evaluation plan for this kind of formula.

A. Properties of Dependent Cycles in Single Component

Theorem 8.1: A recursive formula with a dependent cycle cannot be transformed to a stable formula.

Proof: Case 1: We already proved that a pure multidirectional cycle cannot be transformed to a stable formula. Furthermore, any multidirectional cycle cannot be transformed to a stable one. Indeed, any multidirectional cycle has at least one undirected edge (with two nodes, e.g., x_1 and x_2), and x_1 and x_2 are used as the tails of the directed edge. To be stable, the nodes x_1 and x_2 should be disconnected, but will never be disconnected because later resolution graphs are obtained by appending the I-graph to the head of directed edges, and nodes x_1 and x_2 will never be split. Therefore, a dependent cycle with a multidirectional cycle as a subcycle or a dependent cycle with an undirected edge, whose two nodes are used as the tail of the directed edges is not transformable to a stable one.

Case 2: The same claim can be applied to a component with one undirected edge whose two nodes are used as the heads of the directed edges. On the next expansion, the two nodes of the undirected edge become the tails of the two directed edges, and from Case 1 we can easily find that the component cannot be transformed to a stable one.

Case 3 Let us consider the dependent, one-directional cycle. Assume that there is an extra undirected edge that makes the cycle dependent. We can assume that the node (call the node x) of the extra undirected edge is used as a tail of one directed edge, and the other node (called y) is used as a head of one directed edge because we already mentioned the other possible cases. If the value of x is given on a query (only the value of one variable is known), then two variables are

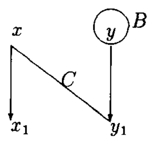


Fig. 15. I-graph for (r.7.1).

determined in the next expansion. On further expansions, there is no possibility that only one variable is determined. So the formula cannot be transformed to a stable formula. Q.E.D.

Corollary 8.1: A formula can be transformed to an equivalent strongly stable formula if and only if it has only pure one-directional cycles.

Proof: From the previous theorems. Q.E.D.

Remark: We have shown that the "*semantic*" and the "*syntactic*" definitions of strongly stable and transformable to strongly stable formulas are equivalent. Therefore, only one-directional cycles can be transformed to stable formulas.

B. Query Evaluations

We begin with several examples for illustrating the use of the resolution graph in recursive formulas with dependent cycles.

Example 8.1: Consider the following recursive formula.

$$P(x,y) : -A(x,x_1) \wedge B(y,y_1) \wedge C(x_1,y_1) \wedge P(x_1,y_1). \quad \text{(r.8.1)}$$

The corresponding first and second resolution graphs for the recursive rule (r.8.1) are in Fig. 16. If a query form $? - P(d,v)$ is given, from the second expansion, all the variables in the recursive predicate are determined, and there is no nondetermined part. The query evaluation plan for $? - P(d,v)$ is the following: σE, $\sigma A - C - B - E$, $\sigma A - C - B - \{{A \atop B}\} - C - E$. (The symbol $\{\}$ is used to express the parallel evaluation of relations.) The simplified formula is σE, $\bigcup_{k=0}^{\infty} \sigma A - C - B - [\{{A \atop B}\} - C]^k - E$. Note that the evaluation plan form is similar to transitive closure formula because, after certain expansion, since two cycles are connected, we can consider two variables as one vector and then the rule can be thought of as a pure unit cycle that produce a transitive closure formula. Q.E.D.

Example 8.2: Consider the following recursive formula.

$$P(x,y,z) : -A(x,z_1) \wedge B(x_1,y) \wedge C(y_1,z) \wedge D(y_1,z_1) \wedge P(x_1,y_1,z_1). \quad \text{(r.8.2)}$$

The corresponding I-graph is in Fig. 17. If a query $? - P(d,v,v)$ is given, on the next expansion we have $P(v,d,d)$, and on third expansion, we have $P(d,d,d)$. From the fourth expansion on, all the variables are determined, and data retrieval in further expansions can be minimized. We also can apply parallel evaluation to reduce the query processing time as much as possible. The query evaluation plan for the query $? - P(d,v,v)$ is shown in Fig. 18. Q.E.D.

There are many possible connections among cycles and acyclic paths. Two or more cycles (each could be a unit cycle, one-directional cycle, or multidirectional cycle) can be connected by one or more undirected edges, directed edges, or a mixture of both. An acyclic path can be connected to a pure cycle or a connected graph of cycle(s). Here, we will consider one example: two unit, rotational cycles connected by one undirected edge. However, we should note that the theorem derived from this example cannot be generalized to all the cases of complicated connections.

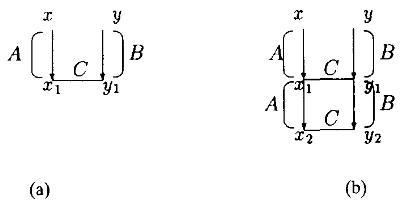


Fig. 16. I-graph for (r.8.1).

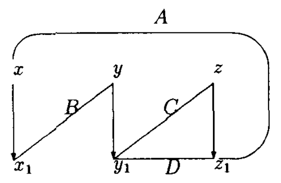


Fig. 17. I-graph for (r.8.2).

$$ADCEB, \ \bigcup_{k=0}^{\infty} ADC - CDAB - B - \left[\left\{ \begin{matrix} A \\ B \\ C \end{matrix} \right\} - D \right]^k - E$$

Fig. 18. Evaluation plan for a query to (r.8.2).

There are four possible cases of connecting two unit, rotational cycles with an undirected edge as in rules (r.8.3) to (r.8.6) [Fig. 19(a)–(d)].

Example 8.3:

$$P(x,y) : -A(x,u) \wedge B(y,v) \wedge C(x,y) \wedge P(u,v) \quad \text{(r.8.3)}$$
$$P(x,y) : -A(x,u) \wedge B(y,v) \wedge C(u,v) \wedge P(u,v) \quad \text{(r.8.4)}$$
$$P(x,y) : -A(x,u) \wedge B(y,v) \wedge C(x,v) \wedge P(u,v) \quad \text{(r.8.5)}$$
$$P(x,y) : -A(x,u) \wedge B(y,v) \wedge C(u,y) \wedge P(u,v). \quad \text{(r.8.6)}$$

The undirected edge C across two cycles indicates that a join operation can be applied to the undirected edges A and B to group them together, and the rule becomes $P(x,y) : -D(x,y,u,v) \wedge P(u,v)$ where $D(x,y,u,v) : -A(x,u) \wedge B(y,v) \wedge C(x,y)$ in the rule (r.8.3), $D(x,y,u,v) : -A(x,u) \wedge B(y,v) \wedge C(u,v)$ in the rule (r.8.4), $D(x,y,u,v) : -A(x,u) \wedge B(y,v) \wedge C(x,v)$ in the rule (r.8.5), $D(x,y,u,v) : -A(x,u) \wedge B(y,v) \wedge C(u,y)$ in the rule (r.8.3).

If we view x and y as a vector z and u and v as a vector w, the rule becomes $P(z) : -D(z,w) \wedge P(w)$ whose compiled formula is D^*E which can be processed using the transitive closure query processing strategies. Although we take two variables x and y as one vector, the evaluation of the vector should be treated carefully. Different variables within one vector may be instantiated and inquired differently in queries.

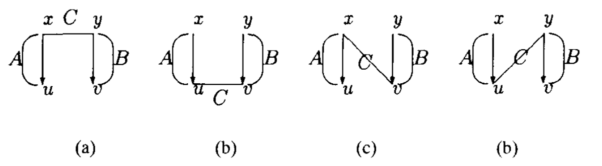


Fig. 19. I-graphs for (r.8.3), (r.8.4), (r.8.5), and (r.8.6).

For example, in query $? - P(a, y)$, the two variables in the vector z are instantiated differently. Q.E.D.

Other connections, including multiple connections, can be handled similarly. Hence, we have the following theorem.

Theorem 8.2: If the I-graph of a linear recursive formula contains undirected edge connections between unit cycles, the connected variables can be collapsed into one vector in compilation.

Proof: The proof can be derived from the study of all possible cases. The above example has examined the case of two unit cycles connected by one undirected edge. Obviously, two unit cycles can also be connected by two or more undirected edges, which makes the linkage between two cycles even stronger than one undirected edge. Thus, they should be treated as joining together and merging two cycles into one by vectoring the variables. In the cases that more than two unit cycles are connected by undirected edges, the discussion can be generalized similarly. In general, an n-D unit cycle formula is reduced to an m-D unit cycle recursion by collapsing two or more unit cycles. Q.E.D.

IX. Heterogeneous Components (4)

In this section, we will consider recursive formulas having I-graphs with multiple components which are composed of disjoint combinations of different classes. The general method for this class is not known at this time. Further studies should be done on the recursive formulas in this class. Using the properties of each of the classes we discussed so far, we can derive the compiled formula and the query evaluation plan for particular cases.

A. Closure Properties of Recursive Formulas

Theorem 9.1: A recursive formula with a one-directional permutational cycle or disjoint combinations of such cycles is also permutational.

Proof: A disjoint combination of permutational cycles is also permutational because there are no new variables in the recursive predicates. As soon as the formula becomes stable, it is in fact in its original form. Q.E.D.

Example 9.1: Consider the following recursive formula.

$$P(x, y, z, u, v, w) : -P(z, y, u, x, w, v). \qquad \text{(r.9.1)}$$

The recursive rule (r.9.1) is a permutational formula, and there are three permutational cycles in the I-graph with weights 3, 1, and 2, respectively. We can easily see that the formula becomes stable (comes back to the original formula) after six expansions and will not produce new tuples by further expansions, therefore further expansions are meaningless. Q.E.D.

Theorem 9.2: A formula constructed by a disjoint combination of one or more one-directional cycles can be transformed to an equivalent strongly stable formula.

Proof: If there are k disjoint independent, one-directional cycles $G_1, G_2, \cdots G_k$ from a recursive formula, and the weight of the cycle for each G_i is c_i, the formula can be transformed to an equivalent stable formula by unfolding exactly L times, where L is the least common multiple of $c_1, c_2, \cdots c_k$.Q.E.D.

Example 9.2: Consider another recursive formula.

$$P(x, y, z, u, w, s, v) : -A(x, t) \wedge P(t, z, y, w, s, r, v) \wedge B(u, r). \qquad \text{(r.9.2)}$$

The corresponding I-graph has 4 one-directional disjoint cycles of weights 1, 2, 3, and 1, respectively. We can easily see that the formula becomes stable after six expansions. Q.E.D.

We should point out that there are other recursive formulas which may stabilize for particular queries but are not strongly stable and are not equivalent to a strongly stable formula.

Theorem 9.3: A formula constructed by a disjoint combination of stable cycles (one-directional cycles) and nontransformable cycles (at least one nontransformable cycle) is not transformed to a stable formula.

Proof: All the disjoint components will be expanded independently (not connected), therefore we can see the property easily. Q.E.D.

From the classification, we can find the following properties of recursive formulas. Recall the various cases.

Theorem 9.4: A formula constructed by a disjoint combination of classes {acyclic paths (1), pure one-directional unit permutational cycles (2.1.2), pure one-directional nonunit permutational cycles (2.1.4), pure multidirectional bounded cycles [a subset of (2.2)] } is bounded.

Proof: Each component will be expanded independently, and all the components are bounded, therefore the formula is bounded. Q.E.D.

Remark: A formula constructed by a disjoint combination of bounded components is bounded.

Theorem 9.5: If a formula P is constructed by a disjoint combination of classes {pure one-directional unit permutational cycles (2.1.2), pure one-directional nonunit permutational cycles (2.1.4)}, then the tight upper bound of P is the least common multiple (L) -1 of the weights of all the cycles.

Proof: After L expansions, the formula comes back to the original form. Therefore, the formula is bounded and the upper bound is $L - 1$. Q.E.D.

Now we can conclude the following theorem.

Theorem 9.6: The above classification is complete.

Proof: Our analysis is done on each component. There are four possibilities on each component, 1) no nontrivial cycle, 2) pure one-directional cycles, 3) pure multidirectional cycles, 4) dependent cycles. There is no overlap between each pair of these classes. A disjoint combination of cycles in the same class will be in the same class. A disjoint combination of the different classes will be in the heterogeneous components class. Therefore, the classification is complete. Q.E.D.

X. Discussion

We discuss a number of issues which are closely related to our methodology. First, we discuss the issue of relaxing the soft assumptions that were imposed in Section II. Second, we discuss the use of equality (EQ) predicate in handling some complex linear recursive formulas. Third, we discuss the possibility of applying our graph model to the class of nonlinear recursive formulas.

A. Relaxing Soft Assumptions

We can relax two soft assumptions in Section II which were imposed on the target class of linear recursive formulas in our research.

Relaxing No-Constant Symbol Assumption: We assumed that there were no constant symbols in the statement. Constants were introduced by the given query. The reason for this assumption was for generality. Variables in the statement can always be replaced by constants, and a statement with constants in it can be considered as an instance of the formula which contains only variables.

If there are constants in the nonrecursive predicate, regardless of the query form, we can apply selection operations using those constant symbols.

Example 10.1: Consider the following recursive formula having a constant symbol (here, 'b').

$$P(x,y) : -A(x,b,z) \wedge P(z,w) \wedge B(w,y).$$

If a query $? - P(a,y)$ is given, the selection operation on relation A is more restrictive, i.e., finding all the tuples that satisfy $x = a$ and $y = b$ from the relation $A(x,y,z)$ is more restrictive than $x = a$. If a query $? - P(x,c)$ is given, we can apply selection operations from relation A and relation B simultaneously and the results can be merged with the exit relation E. The evaluation can be expressed as shown in Fig. 20. Q.E.D.

If there are constants under the recursive predicate in the head, the query form should match with the formula. To satisfy this condition, the corresponding position of the constant symbols from the query should be a variable or exactly the same constant. If there are constants under the recursive predicate in the body, those constant symbols can be used as valuable information from the second expansion for evaluating queries.

If there are constant(s) in the exit rule, for each expansion, the recursive predicate in the body of the recursive rule will be matched with the head of the exit rule, and the situation is similar to the case that allows constant symbols in the nonrecursive predicate.

Therefore, if there are constant symbols in the formula, this will not cause any difficulty for our graph analysis. The only difference is that we can use the information of constant symbols and can make more efficient query evaluation plans.

Relaxing Single Recursion Assumption: We have assumed there was only one recursive statement (single recursion) and one or more nonrecursive statements (exit rules) for a given recursive predicate to make our discussion simple. If there are two or more recursive formulas for a given recursive predicate (multiple recursion), the evaluation strategy for a query becomes more complicated.

$$\begin{tikzcd}
 & \sigma A^k \arrow[dr, dash] & \\
\bigcup_{k=0}^{\infty} & & E \\
 & \sigma B^k \arrow[ur, dash] &
\end{tikzcd}$$

Fig. 20. Evaluation plan for a query in Example 10.1.

Example 10.2: Consider the following recursive formulas.

$$P(x,y) : -A(x,z) \wedge P(z,w) \wedge B(w,y) \qquad \text{(r.10.1)}$$
$$P(x,y) : -C(x,z) \wedge P(z,w) \wedge D(w,y). \qquad \text{(r.10.2)}$$

The recursive formulas (r.10.1) and (r.10.2) are both strongly stable formulas. If a query $? - P(a,y)$ is given, the evaluation plan will be: σE, $\sigma AEB, \sigma CED$, $\sigma AAEBB, \sigma ACEDB, \sigma CCEDD, \sigma CAEBD, \cdots$, and so on. Therefore, the evaluation plan can be written as $\bigcup_{k=0}^{\infty} \sigma R_1 R_2 \ldots R_k E S_k \cdots S_2 S_1$ where R_i is relation A and S_i is relation B, or R_i is relation C and S_i is relation D.Q.E.D.

If there are m distinct recursive formulas for a recursive predicate, for a given query, there are m distinct evaluations by the first expansion, m^2 distinct evaluations by the second expansion, m^3 distinct evaluations by the third expansion, and so on. Therefore, the total number of distinct evaluations for up to k expansions will be: $(m^{k+1} - 1)/(m - 1)$ contrast to k distinct evaluations if there is only one recursive formula for a given recursive predicate. The complexity of the multiple recursion is exponential in the level of expansion k.

If there are two recursive formulas which belong to two different classes, the query evaluation plans become more complicated.

B. Using the EQ Predicate in the Graph Model

Since we did not allow arithmetic predicates in Section II, the equality predicate (EQ) was also discarded.

Using the EQ predicate can cause difficulties. However, if we use the EQ predicate in a restricted manner, it is very useful in handling complex linear recursive formulas. Our approach is to treat the EQ predicate as a special predicate which has value "TRUE" if its two arguments have same value, otherwise, it returns "FALSE." We can treat the predicate EQ as an EDB relation, but the tuples of the relation are not stored explicitly. We should point out that the predicate EQ can be evaluated only if at least one of its two arguments is determined.

Handling Recursive Variables with Constant Symbols: By introducing the new predicate EQ for the constant symbols under the recursive predicate in the head, our graph model can cover the recursive predicate with constant symbols. For example, $P(a,y) : -A(u,z) \wedge P(z,w) \wedge B(w,y)$ can be rewritten using the predicate $'EQ'$ as $P(x,y) : -EQ(x,a) \wedge A(u,z) \wedge P(z,w) \wedge B(w,y)$.

Handing Recursive Formulas with Repeated Variables: If there are variables appearing more than once under the recursive predicate, in general, the kth I-graph is not isomorphic to the original I-graph. Hence, the query evaluation plans using the graph model become more complicated and more difficult to analyze. However, if we use EQ properly, we can easily cope with recursive formulas with repeated variables.

Example 10.3: Consider the following recursive variable with repeated variable.

$$P(x,x) : -A(x,y) \wedge P(y,z) \wedge B(z,x). \qquad \text{(r.10.3)}$$

The recursive formula (r.10.3) can be rewritten as the formula formula (r.10.4) by introducing EQ.

$$P(x,w) : -EQ(w,x) \wedge A(x,y) \wedge P(y,z) \wedge B(z,x). \quad \text{(r.10.4)}$$

The corresponding I-graphs are in Fig. 21(a) and (b), respectively. Q.E.D.

As we can see from the recursive formula (r.10.4), there are no repeated variables in the recursive predicate after introducing the predicate EQ. Therefore, we can apply our graph model to those formulas for classification and query evaluations. The recursive formula F' has the following property.

Theorem 10.1: A recursive formula F' derived from a formula with repeated variables in the recursive predicate is not a strongly stable formula nor can be transformed into a strongly stable one.

Proof: The recursive formula F' has at least one undirected edge labeled "EQ" (with two nodes, e.g., x_1 and x_2), and x_1 and x_2 are used as the tails (or heads) of the directed edge. Therefore, the recursive formula F' has multidirectional cycles or acyclic path(s). We already proved that a multidirectional cycle or an acyclic path cannot be transformed to a strongly stable formula. Q.E.D.

C. Applying the Graph Model to Nonlinear Recursive Formulas

In this section, we will consider formulas with more than one recursive predicate in the body. For example, $P(x,y) : -A(x,w) \wedge P(w,v) \wedge B(v,u) \wedge P(u,z) \wedge C(z,y)$ is a nonlinear recursive formula.

To the best of our knowledge, general compilation techniques for nonlinear recursive formulas have not been proposed in the literature. If we can transform nonlinear recursive formulas into a finite set of linear recursive formulas, we can simply apply the evaluation techniques we have discussed to the nonlinear recursive formulas. It is known that only a subset of nonlinear recursive formulas can be transformed into a finite set of linear recursive formulas [6], [21], [22].

We also do not have complete results on nonlinear recursive formulas. Here we introduce some of our preliminary observations to nonlinear recursive formulas.

Definition 10.1: The *degree* of recursive formula is the number of recursive predicates in the antecedent. Linear recursive formulas are the first degree recursive formulas.

Definition 10.2: A nonlinear recursive formula is *strongly stable* if determined variables of the recursive predicate in the consequent and of all the recursive predicates in the antecedent occur in the same position for any query.

From a recursive formula F with degree k, we can get k linear recursive statements $R_1, R_2, \cdots, R_k$ that contain only one recursive predicate in the antecedent by replacing $k-1$ recursive predicates using exit rules and leaving one recursive predicate. For convenience, we can assume that R_1 keeps the first recursive predicate P_1, R_2 keeps the second recursive predicate P_2, and so on. The formula R_i is the *ith simplified linear recursive formula of F*. Then we have the following theorem.

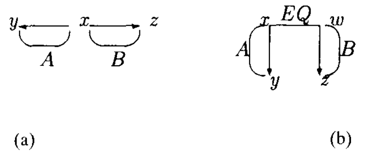


Fig. 21. I-graphs for (r.10.3) and (r.10.4).

Theorem 10.2: A recursive formula F with degree k is strongly stable if and only if each of the recursive formulas $R_1, R_2, \cdots, R_k$ is a strongly stable formula.

Proof: (Only If): If any of R_i's is not strongly stable, the formula F is not a strongly stable formula when recursive predicate P_i is encountered for expansion. Therefore, all R_i's should be strongly stable formulas.

(If): If all the R_i's are strongly stable formulas, each expansion of P_i will preserve the property of strongly stable and all the expanded formulas are strongly stable formulas. Therefore, formula F is a strongly stable formula. Q.E.D.

Example 10.5: Consider the following nonlinear recursive formula.

$$P(x,y) : -A(x,w) \wedge P(w,v) \wedge B(v,u) \wedge P(u,z) \wedge C(z,y). \qquad \text{(r.10.5)}$$

To study nonlinear recursive formulas F, we will analyze the corresponding simplified linear recursive formulas of F. The first and second simplified recursive formula of (r.10.5) are (r.10.6) and (r.10.7), respectively.

$$P(x,y) : -A(x,w) \wedge P(w,v) \wedge B(v,u) \wedge E(u,z) \wedge C(z,y) \qquad \text{(r.10.6)}$$

$$P(x,y) : -A(x,w) \wedge E(w,v) \wedge B(v,u) \wedge P(u,z) \wedge C(z,y). \qquad \text{(r.10.7)}$$

The corresponding I-graphs for recursive formulas (r.10.6) and (r.10.7) are in Fig. 22(a) and (b), respectively. We can easily find that the two simplified linear recursive formulas of (r.10.5) are strongly stable and hence the recursive formula (r.10.5) is strongly stable. Q.E.D.

Example 10.5: Let us consider recursive formulas that are not strongly stable. For example,

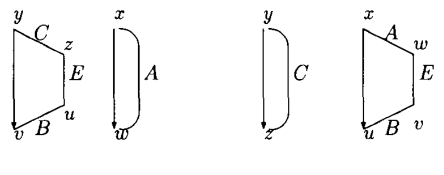
$$P(x,y) : -A(x,z) \wedge P(z,w) \wedge B(w,u) \wedge P(v,u) \wedge C(v,y). \qquad \text{(r.10.8)}$$

The first and second simplified recursive formulas of (r.10.8) are in (r.10.9) and (r.10.10), respectively.

$$P(x,y) : -A(x,z) \wedge P(z,w) \wedge B(w,u) \wedge E(v,u) \wedge C(v,y) \qquad \text{(r.10.9)}$$

$$P(x,y) : -A(x,z) \wedge E(z,w) \wedge B(w,u) \wedge P(v,u) \wedge C(v,y). \qquad \text{(r.10.10)}$$

The corresponding I-graphs are in Fig. 23. As we can see from Fig. 23, the recursive formula (r.10.9) is strongly stable, but the recursive formula (r.10.10) is not strongly stable. Let us



(a) (b)

Fig. 22. I-graphs for (r.10.6) and (r.10.7).

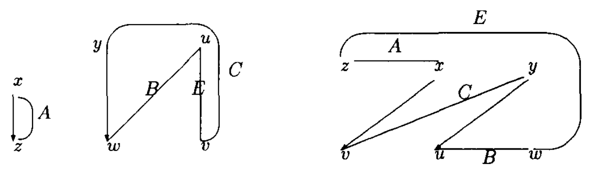


Fig. 23. I-graphs for (r.10.9) and (r.10.10).

consider the evaluation plan for the query $?-P(c,X)$. Let us assume that the evaluation plan for the recursive predicate $P(c,X)$ is G, and the evaluation plan for the recursive predicate $P(X,c)$ is H, where c is a generic constant and X is a variable. Then the formula (r.10.8) will be expressed in a simplified form as

$$\begin{aligned} &G:-A,G,B,H,C\\ &H:-C,G,B,H,A\\ &G:-E\\ &H:-E \end{aligned}$$

where the order of the predicates in the antecedent expresses the order of evaluation because there are common variables between adjacent predicates so that join operation can be applied, i.e., the evaluation of G will be done by applying selection operation to the predicate A, then join with G where G can be expanded again or can be replaced by the exit rule E, then the union of intermediate results from G will be joined with relation B, the join operation will be applied with recursive evaluation predicate H, where H can be expanded further or can be replaced by the predicate E, and finally join operation will be applied to predicate C to derive answers.

The recursive expansion of G and H cannot be expressed by direct recursion, i.e., G cannot be expressed in terms of G only (with no H involved) and H cannot be expressed in terms of H (with no G involved). We call this recursive relation, *irreducible mutual recursion*. Q.E.D.

We hope a more complete study of nonlinear recursive formulas will be forthcoming.

XI. Conclusions

We presented a classification and compilation scheme of linear recursive formulas based on a graph model. Our analysis of the compilation and query processing of linear recursive formulas discloses that the formulas in each class share common characteristics in their compiled formulas and query processing plans. Therefore, it shows that the I-graph model is a valuable tool in the classification of recursive formulas and the resolution graph is a powerful tool in deriving planning mechanisms for recursive queries.

We believe that our graph method provides a powerful tool in the study of the behavior of linear recursive formulas with complex variable patterns as well as linear recursive formulas with simple variable patterns. Once such behavior is well understood, the compilation and further optimization can be explored in depth, which in turn will have strong impact on the further development of efficient recursive query processing methods.

Although we showed some fundamental results on unbounded cycles, dependent cycles, and heterogeneous components, further exploration will produce interesting results. Moreover, the exploration of the application of our graph model to the compilation of multiple linear recursive rules, nonlinear recursive rules, and other kinds of recursion is another future research topic.

Acknowledgment

We gratefully thank an anonymous referee for his insightful comments on an earlier version of this paper.

References

[1] F. Bancilhon and P. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," in *Proc ACM-SIGMOD Conf. Management of Data*, 1986, pp. 16–52.
[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman, "Magic sets and other strange ways to implement logic programs," in *Proc. 5th ACM SIGMOD-SIGART Symp. Principles of Database Syst.*, 1986, pp. 1–15.
[3] A. Chandra, "Programming primitives for database languages," in *Proc. 8th Annu. ACM Symp. Principles of Programming Languages*, 1981.
[4] H. Gallaire, J. Minker, and J. Nicolas, "Logic and databases: A deductive approach," *Comput. Surveys*, vol. 16, no. 2, pp. 153–185, June 1984.
[5] J. Han and L. J. Henschen, "Handling redundancy in the processing of recursive database queries," in *Proc. ACM-SIGMOD Conf. Management of Data*, San Francisco, CA, June 1987, pp. 73–81.
[6] J. Han, "Pattern-based and knowledge-directed query compilation for recursive databases," Comput. Sci. Tech. Rep. 629, Ph.D. dissertation, Univ. of Wisconsin, Madison, Dec. 1985.
[7] J. Han, L. Henschen, and C. Youn, "Compiling complex linear recursive cluster," in *Proc. 1988 Canadian Inform. Processing Society Conf.*, Edmonton, Alta., Canada, 1988.
[8] L. J. Henschen and S. Naqvi, "On compiling queries in recursive first-order databases," *J. ACM* vol. 31, no. 1, pp. 47–85, 1984.
[9] Y. Ioannidis, "A time bound on the materialization of some recursively defined views," *Proc. 11th Int. Conf. Very Large Databases*, Stockholm, Sweden, Aug. 1985, pp. 219–226.
[10] ——, "Bounded recursion in deductive databases," Memorandum UCB/ERL M85/6, Univ. of California, Berkeley, 1985.
[11] H. V. Jagadish, R. Agrawal, and L. Ness, "A study of transtive closure as a recursion mechanism," *Proc. ACM-SIGMOD Conf. Management of Data*, San Francisco, CA, June 1987, pp. 331–344.
[12] J. W. Lloyd, *Foundations of Logic Programming*, second extended ed. New York: Springer-Verlag, 1987.
[13] D. Maier, *The Theory of Relational Databases.* Rockville, MD: Computer Science Press, 1983.
[14] J. Minker, "Search strategy and selection function for an inferential relational systems," *ACM Trans. Database Syst.*, vol. 3, no. 1, 1978.
[15] ——, *Foundations of Deductive Databases and Logic Programming*, (edited). Los Altos, CA: Morgan-Kaufmann, 1988
[16] J. Naughton, "Data independent recursion in deductive databases," in *Proc. ACM SIGACT-SIGMOD Symp. Principles of Database Syst.*, 1986.
[17] J. Naughton, "One-sided recursions," in *Proc Sixth ACM SIGACT-SIGMOD Symp. Principles of Database Syst.*, 1987, pp. 340–348.
[18] ——, "Minimizing function-free recursive inference rules," *J. ACM*, vol. 36, no. 1, pp. 69–91, 1989.
[19] R. Ramakrishnan, private communication, 1988.

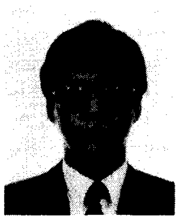
[20] R. Reiter, "On closed world databases," in *Logic and Databases*, H. Galaire and J. Minker, Eds. New York: Plenum, 1978, pp. 55–76.

[21] O. Shmueli, "Decidability and expressiveness aspects of logic queries," in *Proc. 5th ACM SIGMOD-SIGART Symp. Principles of Database Syst.*, 1987.

[22] D. J. Troy, C. T. Yu, and W. Zhang, "Linearization of nonlinear recursive rules," *IEEE Trans. Software Eng.*, vol. 16, no. 9, pp. 1109–1119, 1989.

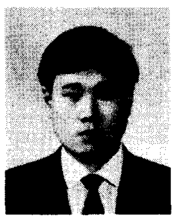
[23] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Vol. II, The New Technologies.* Rockville, MD: Computer Science Press, 1989.

[24] C. Youn, L. Henschen, and J. Han "A classification of recursive formulas in deductive databases," in *Proc. ACM-SIGMOD Conf. Management of Data*, Chicago, IL, June 1988.



Cheong Youn received the B.S. degree in physics from Seoul National University, Seoul, Korea, in 1979, the M.A. degree in computer science from Sangamon State University, Springfield, IL, in 1983, and the Ph.D. degree in computer science from Northwestern University, Evanston, IL, in 1988.

Since 1988, he has been with Bell Communications Research where he is currently a Member of Technical Staff in the Planning and Engineering Laboratory. His research interests include information modeling, software engineering, deductive databases, and object-oriented databases.



Hyoung-Joo Kim (S'82–M'88) received the B.S. degree in computer engineering from Seoul National University, Seoul, Korea, in 1982, and the M.S. and Ph.D. degrees in computer science from the University of Texas at Austin in 1985 and 1988, respectively.

He is currently an Assistant Professor of Computer Engineering, Seoul National University, Seoul, Korea. He has been on the faculty of the College of Computing at the Georgia Institute of Technology between 1988 and 1990. His current research interests include object-oriented systems, software development environments, graphical environments, and parallel computing.

Dr. Kim is a member of the Association for Computing Machinery and the IEEE Computer Society.

Lawrence J. Henschen received the B.A., M.A., and Ph.D. degrees all from the University of Illinois at Urbana.

He is a Professor of Electrical Engineering and Computer Science, Northwestern University, since 1971. He is the co-author of over 70 journal and conference papers. He developed with L. Wos many early theoretical about Horn clauses upon which much of Prolog is based. He pioneered the compilation approach for recursive databases. He continues to develop research in automated reasoning and its application to intelligent databases.

Jiawei Han received the M.Sc. degree in 1981 and the Ph.D. degree in 1985, both in computer sciences from the University of Wisconsin, Madison.

He was an Assistant Professor of Northwestern University from 1986 to 1987. Currently, he is an Assistant Professor of Simon Fraser University, Burnaby, B.C., Canada. His current research interests include database systems, deductive database systems, logic programming, knowledge discovery in databases, and artificial intelligence.

Dr. Han is a member of the Association for Computing Machinery, the Association for Logic Programming, and the IEEE Computer Society.