An Efficient Algorithm for the Generalized Partially Instantiated Same Generation Query in Deductive Databases

Nabil Arman
Palestine Polytechnic University, Palestine

Abstract: *The expressive power and intelligence of traditional database systems can be improved by recursion. Using recursion, relational database systems are extended into knowledge-base systems (deductive database systems). Linear recursion is the most frequently found type of recursion in deductive databases. In this paper, an algorithm to solve the generalized partially instantiated form of the same generation query in deductive databases is presented. The algorithm uses special data structures, namely, a special matrix that stores paths from roots of the graph representing a two-attribute normalized database relation to all nodes reachable from these roots, and a reverse matrix that stores paths from any node to all roots related to that node. Using simulation, this paper also studies the performance of the algorithm and compares that with the standard depth-first search based algorithms.*

Keywords: *Deductive databases, linear recursive rules, same generation query.*

Received April 28, 2003; accepted July 22, 2003

1. Introduction

The development of efficient algorithms to process recursive rules and queries within the context of large database systems has recently attracted a large amount of research efforts due to the important role of recursive rules in improving the intelligence of database systems and extending them into knowledge-base systems [1-11]. One of the main features of these intelligent database systems, namely deductive databases, is their ability to define recursive rules and to process queries on them directly.

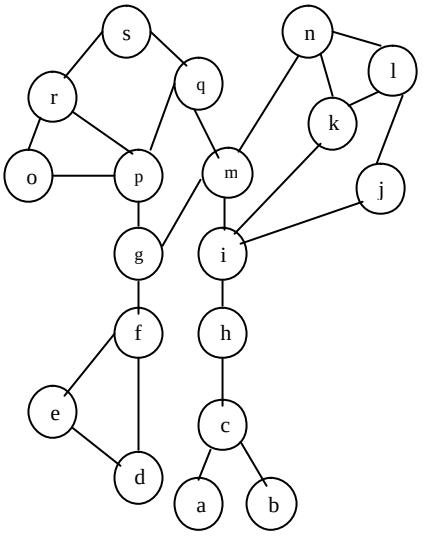
In deductive databases, most recursive rules appear in a simple form in which the rule's head appears only once in the body of the rule [5]. In general, this type of logic rules is called *linearly recursive*. A same generation (*sg*) rule is a linearly recursive rule of the following form:

$sg(X_1, X_2, ..., X_n)$:- $par(Y_1, X_1)$, $par(Y_2, X_2)$,…, $par(Y_n, X_n)$, $sg(Y_1, Y_2, ..., Y_n)$

where "*par*" is an extensional (base) predicate and "*sg*" is an intentional database predicate. Within the context of deductive databases, the extensional database predicate "*par*" is defined by a two-attribute normalized database relation with very many tuples as shown in Figure 1-a [5, 7]. Another common view for the base relation is represented by a directed graph, as shown in Figure 1-b. For every tuple <*x*,*y*> of the base relation, there exists, in the corresponding graph, a directed edge from node *x* to node *y*. The nodes in such a graph are the set of distinct values in the two columns of the base relation (i.e., the domain).

X	Y
s	r
s	q
r	o
r	p
q	p
q	m
p	o
p	g
n	m
n	k
m	l
m	g
l	i
k	j
j	i
i	h
h	c
g	f
f	e
f	d
e	d
c	a
c	b

a) Relation form



b) Graph form

Figure 1. The binary relation "par" in a) table form b) graph form.

To generate solutions from the above recursive rule, another non-recursive rule, the exit rule, which defines the predicate "$sg(X_1, X_2, ..., X_n)$" must exist. This non-recursive rule is given by:

$sg(X_1, X_2, ..., X_n)$:- $par(Y, X_1)$, $par(Y, X_2)$,…, $par(Y, X_n)$

A query on a predicate that is defined by the the recursive and the exit rule is called a same generation query. This query is a headless rule of the following form:

$$:\text{-}\ sg(X_1, X_2, ..., X_n)$$

A query typically involves a predicate symbol with some variable arguments, and its meaning or answer is the different constant combinations that when bound (assigned) to the variables, can make the predicate true. In general, an *n*-place unit query, such as the above one, may have different forms depending on the instantiation status of the variables [8]. In this article, we propose an algorithm for solving the generalized partially instantiated form of the same generation query, i.e., a query that has the form:

$$:\text{-}\ sg(X_1, X_2, ..., X_i, c_{i+1}, c_{i+2}, ..., c_n)$$

where $X_1, X_2, ..., X_i$ are the uninstantiated variables whose values are to be determined and $c_{i+1}, c_{i+2}, ..., c_n$ are constants representing nodes in the graph. The order of the arguments is irrelevant since "*sg*" is a symmetric relation. Let the uninstantiated set of nodes (USN) be $\{X_1, X_2, ..., X_i\}$ and the instantiated set of nodes (ISN) be $\{c_{i+1}, c_{i+2}, ..., c_n\}$, then the answer of such a query is the set of nodes with a cardinality of *i* that are of the same generation as $c_{i+1}, c_{i+2}, ..., c_n$ (i.e., the set of nodes that are on the same level of a family tree with $c_{i+1}, c_{i+2}, ..., c_n$).

The counting technique for linear rules and the magic-sets rule rewriting are the two best-known techniques to solve such query in its simplest form (without the generalization to *n*-place queries) [9]. In solving queries like the partially instantiated same generation, these techniques process every node on the paths of the graph. The Modified HaNa method is similar to the counting technique and the magic-sets rule rewriting in the sense that it processes all the nodes on the paths of the graph [6]. The generalized partially instantiated same generation query algorithm, as presented in this article, is more efficient than these techniques. The algorithm considers the relevant part of the graph/database by examining the set of nodes that are connected to the selected node from the set of instantiated nodes in the query and involves less computation.

2. The Structure Used in the Algorithm

The structure used in the algorithm is a special matrix. This structure has been used in computing the transitive closure of a database relation [8], and in answering the simple form of the partially instantiated same generation query in deductive databases [7]. In this matrix, the rows represent some paths in the graph starting from the roots/source nodes to the leaves. Basically, depth-first search is used to create the paths of the graph. Instead of storing every node in all paths, the common parts of these paths can be stored only once to avoid duplications. If two paths $P_1 = <a_1, a_2 ... a_n, b_1, b_2, ... b_m>$ and $P_2 = <a_1, a_2, ... a_n, c_1, c_2, ... c_l>$ have the common parts $<a_1, a_2, ..., a_n>$, then P_1 and P_2 can be stored in the two consecutive rows of the matrix as $< a_1, a_2, ..., a_n, b_1, b_2, ..., b_m >$ and $<$ -- *n empty entries* -- $, c_1, c_2, ..., c_l >$, where the first *n* entries of the second row are empty. To prevent the duplicate storage of the nodes in the matrix, a different technique is used; for the first visit to the node, it is entered into the matrix and the coordinates of its location is recorded. On subsequent visits, instead of entering the node itself, its coordinates are entered into the matrix (a pointer to the already stored node). In this way, only a single copy of each of the graph's nodes is guaranteed to be entered in the matrix. Moreover, there will be only one entry (either a node or a pointer) in the matrix for each edge in the graph. In Figure 2-a, the matrix representation of the graph given in Figure 1-b is presented. In that graph, there are 25 edges, and in its matrix representation there are 25+2 =27 nonempty entries in the matrix (another two entries for the nodes *s* and *n*). An important advantage of this matrix structure is that it stores a path from each node to all the roots that can reach the node.

	0	1	2	3	4	5	6
0	s	r	o				
1			p	0,2			
2				g	f	e	d
3						2,6	
4		q	1,2				
5			m	2,3			
6				i	h	c	a
7							b
8	n	5,2					
9		k	6,3				
10		l	9,1				
11			j	6,3			

a) Matrix representation.

	0	1	2	3	4	5	6
0	o	r	s				
1		p	0,1				
2			q	0,2			
3	d	e	f	g	1,1		
4					m	2,2	
5						n	
6		3,2					
7	a	c	h	i	4,4		
8					k	5,5	
9						1	5,5
10					j	9,5	
11	b	7,1					

b) Reverse matrix representation.

Figure 2. Matrix and reverse matrix representation of the graph of Figure 1.

0,0	0,1	0,2	1,2	1,3	2,3	2,4	2,5	2,6	3,5	4,1	4,2	5,2	5,3	6,3
s	r	o	p	0,2	g	f	e	d	2,6	q	1,2	m	2,3	i

6,4	6,5	6,6	7,6	8,0	8,1	9,1	9,2	10,1	10,2	11,2	11,3
h	c	a	b	n	5,2	k	6,3	l	9,1	j	6,3

Figure 3. An array representation of the matrix of Figure 2-a.

In the implementation of this sparse matrix, the empty entries are not stored explicitly. The matrix can be stored sequentially row by row as shown in Figure 3. For each row, storing the column number of its first non-empty entry and the sequence of non-empty entries in the row is sufficient. Thus, the size of the stored matrix is much smaller than the original relation and matrix.

After the special matrix form is created, a (reverse) matrix, which is the matrix representation of the reverse graph, is generated using the reverse graph. For our purposes, the reversed graph is defined as follows.

Definition: Let $G=(V,E)$ be a graph, where V is a finite set of vertices/nodes and E is a finite set of arcs/edges such that each arc e in E is associated with an ordered pair of vertices/nodes v and w, written as $e=(v,w)$, then the reverse graph RG is given by $RG=(V,E')$ where V is a finite set of vertices/nodes (the same set of vertices of the original graph) and E' is a finite set of arcs such that each arc e' in E' is associated with an ordered pair of vertices w and v, written by $e'=(w,v)$ for each $e=(v,w)$ in E.

The reverse matrix representation generated from the graph in Figure 1-b is the matrix given in Figure 2-b. An important advantage of this matrix structure is that it stores paths from every node to the root node(s). For solving the same generation query, we are interested in the parents and ancestors of a certain node and not in its descendants and this information can be extracted easily from the reverse matrix (and not from the original matrix). Therefore, we need the reverse matrix representation. The reverse matrix can also be stored sequentially row by row as explained for the original matrix. In fact, there is no need even to store the whole matrix structure, because storing the row beginnings, row ends, the entries stored at the row ends, and matrix coordinates of the nodes is sufficient. This is due to the fact that we are interested in the path lengths and not in the stored nodes themselves, from the reverse matrix structure.

3. The Generalized Partially Instantiated Same Generation Query Algorithm

As mentioned before, the matrix structure stores paths from the roots to all nodes reachable from these roots. This means that the nodes in the matrix are clustered on the roots of the graph, i.e., starting from any root, all nodes reachable from that root can be accessed. The reverse matrix structure stores paths from each node to all roots related to that node, which means that the nodes in the reverse matrix are clustered on the leaves of the graph, i.e., starting from any node, all roots related to that node can be accessed. This information can be exploited to solve the generalized partially instantiated same generation query. In solving such a query, the algorithm proceeds as follows:

1. Starting from one of the nodes in the instantiated set of nodes (ISN) of the query and using the reverse matrix structure, the path lengths to all relevant roots are determined. During this computation, only the row beginnings and ends are used. In addition to that, only the relevant roots of the graph are considered. After that, these path lengths are sorted in ascending order, according to the roots and lengths, and duplicate paths are removed.
2. Taking each root from the above step and using the forward matrix structure, all nodes having the same path lengths as the selected node from step (1) are determined. Let this set of nodes in the result be (RS). In this step, only the row beginnings and row ends are also used in the computation of the paths. The original matrix entries are used only in the collection of nodes.
3. Having all nodes (RS) collected in step (2), the algorithm makes sure that all nodes in ISN are in the result i.e., ISN $\subseteq$ RS. In addition, the number of nodes in RS-ISN should be greater than or equal to the number of nodes in USN (i.e., $|\text{RS-ISN}| \geq |\text{USN}|$). The result of the query will consist of all combinations of the nodes in the set RS-ISN.

The path lengths will be sorted because the algorithm will collect all the nodes in the same generation with the given node in a single step. For example, if a certain node has a set of path lengths $\{l_1, l_2, ..., l_k \mid l_1 < l_2 < ... < l_k\}$ from the selected query node, then all nodes that are reachable from that root node with these path lengths are collected in a single step. The duplicate paths will be removed because they will not add new nodes to the solution set. The generalized partially instantiated same generation query algorithm, as described above, can be summarized as shown in Figure 4.

It is worth emphasizing that this algorithm considers only the relevant part of the database/graph, i.e., it considers only the set of nodes that are somehow relevant to the instantiated part of the query (the node in ISN that has been used in determining the path lengths to all relevant roots). In addition to that, the algorithm jumps from one node to another, skipping many nodes on the paths of the underlying graph, since

it only uses the row beginnings and row ends of the matrices in the computation of the paths rather than the nodes of the graph themselves.

Procedure Generalized_Partially_Instantiated_Same_Generation()
begin
Compute all paths from one of nodes in ISN to all relevant roots using the reverse matrix structure.
Sort the path lengths in ascending order.
Remove duplicate paths.
for all generated roots r and using the original matrix structure do
Collect the nodes RS that are of a length as one of the path lengths of r.
if $ISN \subseteq RS$ *and* $|RS\text{-}ISN| \geq |USN|$ *then*
the result of the query will consist of all combinations of the nodes in RS-ISN
end

Figure 4. The generalized partially instantiated same generation query algorithm.

Example: For the graph in Figure 1-b, the answer of the query :- $sg(j, X_1, X_2)$ is computed as follows:

1. The algorithm starts from one of the instantiated arguments (i.e., j, where ISN=$\{j\}$ and USN=$\{X_1, X_2\}$) and uses the reverse matrix structure to determine the set of path lengths to all relevant roots. These paths are sorted and duplicates are removed. Thus, this step generates one path of length 2 to root n.
2. From the above step, the algorithm determines that n is the only relevant root (the root s is not considered in the computation). Therefore, the algorithm starts from n and uses the forward matrix structure to determine all nodes with path lengths of 2 from root n. When a node of path length 2 is reached, it is recorded and the search continues until all relevant parts of the graph is traversed up to path lengths of 2 (the search terminates at this point for the current path of the graph since nodes with lengths greater than 2 are irrelevant in answering the query) or until leaves are encountered. The set of nodes in the result is RS=$\{g,i,k,j\}$.
3. Since ISN $\subseteq$ RS *(i.e.,* $\{j\} \subseteq \{g,i,k,j\}$) and |RS-ISN| $\geq$ |USN| (i.e., $|\{g,i,k\}| \geq |\{X_1, X_2\}|$), then the answer of the query is the set of all combinations of $\{g,i,k\}$, which is equal to $\{(g,i),(g,k),(i,k)\}$. Each combination has two nodes since there are two uninstantiated variables in the query.

4. Performance Evaluation of the Algorithm

To determine the performance of the new algorithm, simulations of the algorithm were performed for random database relations with 2000 tuples with 4 different outdegree values from 1 to 4. For more accurate results, the algorithms were executed 5 times for each case and the average was taken. The same generation query algorithm was tested for 50 randomly generated queries. The number of nodes visited to answer these queries was determined for the algorithm and the depth-first search based technique such as the magic-sets rule rewriting technique and the counting technique. These numbers were plotted for different outdegrees of the randomly generated graphs as shown in Figure 5. When the graph obtained from the execution of the algorithms was examined, two things were observed. First, the number of nodes visited in the algorithms (where the row beginnings and row ends of the matrix representation are visited only) is less than the number of nodes visited in the usual way (where all nodes along the paths are visited). Second, increasing the outdegree of the underlying graph, is in favor of the technique used in visiting the nodes in our algorithm. This is due to the fact that larger outdegree values of the underlying graph generate longer paths, which results in skipping larger number of nodes in the graph.

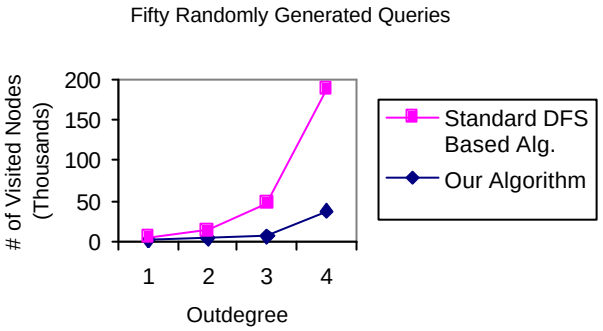


Figure 5. Comparative performance for the generalized partially instantiated same generation query algorithm.

According to [6, 9], the counting technique, the magic-sets rule rewriting, and the Modified HaNa method are not significantly different and they are the best-known techniques to solve such a query. In solving queries like the partially instantiated same generation query, these techniques process every node on the paths of the graph. It is clear that the generalized partially instantiated same generation query algorithm, as presented in this article, is more efficient than the above-mentioned techniques. The algorithm considers the relevant part of the graph/database by examining the set of nodes that are connected to the selected node from the set of instantiated nodes in the query and involves less computation than the above techniques. Therefore, the algorithm solves the generalized partially instantiated same generation query efficiently.

5. Conclusion

This paper presents an efficient algorithm to solve the generalized partially instantiated same generation

query in deductive databases. The algorithm uses special data structures, namely, a matrix representation of the graph, representing the two-attribute normalized database relation, and a reverse matrix representation of the reverse graph. The paper also presents a performance study of the algorithm, and shows the advantages of the techniques used in the algorithm in solving the generalized form of the partially instantiated same generation query in deductive databases. Finally, the paper compares the algorithms with other approaches used to solve such queries like the counting technique, magic-sets rule rewriting, and the Modified HaNa method.

References

[1] Banchilon F., Maire D., Sagiv Y., and Ullman J., "Magic Sets and other Strange Ways to Implement Logic Programs," *in Proceedings of 5th ACM Symp. on Principles of Database System*, pp. 1-15, 1986.

[2] Elmasri R. and Navathe S., *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company Inc., 2000.

[3] Hopfner M., and Seipel D., "Reasoning about Rules in Deductive Databases," *in Proceedings of 17th Workshop on Logic Programming (WLP'2002)*, 2002.

[4] Onet A., "An Approach on Semantic Query Optimization for Deductive Databases," *Informatica*, vol. 48, no. 1, 2003.

[5] Qadah G., Henschen L., and Kim J., "Efficient Algorithms for the Instantiated Transitive Closure Queries," *IEEE Transactions on Software Engineering*, vol. 17, no. 3, 1991.

[6] Suzuki S., Kishi M., and Ibaraki T., "Query Evaluation of the Same Generation Problem with any Variables," *Systms and Computers in Japan*, vol. 24, no. 10, 1993.

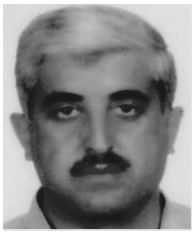
[7] Toroslu I. and Arman N., "An Efficient Algorithm for the Partially Instantiated Same Generation Query in Deductive Databases," *in Proceedings of the 10th International Symposium on Computer and Information Sciences, Istanbul, Turkey*, 1995.

[8] Toroslu I., Qadah G., and Henschen L., "An Efficient Database Transitive Closure Algorithm," *Journal of Applied Intelligence* 4, pp. 205-218, 1994.

[9] Ullman J., Principles of Database and Knowledge-Base Systems, Computer Science Press, 1989.

[10] Yahya A., "Duality for Efficient Query Processing in Disjunctive Deductive Databases," *Journal of Automated Reasoning*, vol. 28, no. 1, pp. 1-34, 2002.

[11] Young C., Kim H., Henschen L., and Han J., "Classification and Compilation of Linear Recursive Queries in Deductive Databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 1, 1992.



Nabil Arman received his BSc in computer science with high honors from Yarmouk University in 1990 and an MSc in computer engineering from Middle East Technical University in 1995. He also received an MSc in computer science from The American University of Washington DC in 1997. In May 2000, he received his PhD in information technology/computer science from the School of Information Technology and Engineering, George Mason University, Virginia, USA. He is an assistant professor at Palestine Polytechnic University, Hebron, Palestine, and was on a one-year leave teaching at the Department of Computer Science, Alisra University while part of this work was done. He is interested in database and knowledge-base systems, and algorithms.