AN EFFICIENT ALGORITHM FOR CHECKING PATH EXISTENCE BETWEEN GRAPH VERTICES

NABIL ARMAN
Associate Professor of Computer Science
Palestine Polytechnic University
Hebron, Palestine

Abstract: The development of efficient algorithms to determine path existence between vertices in a directed graph has attracted a large amount of research efforts due to the important role of these algorithms in many application domains, including deductive databases, rule bases verification, and graph theory to name a few. These algorithms are very crucial to answer certain queries about the path existence between any two graph vertices. This paper presents an efficient algorithm to determine the path existence between graph vertices by utilizing a special matrix representation of the graph.

Keywords: Graph Algorithms, Path Existence Query.

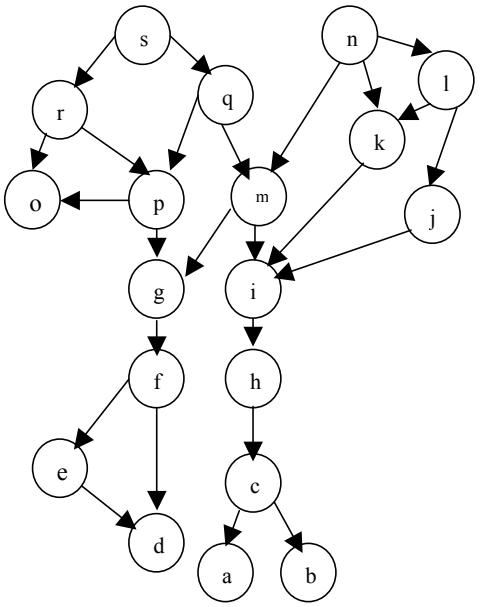
1. INTRODUCTION

Graph algorithms have attracted a large amount of research efforts due to the important role of these algorithms in many application domains. The development of efficient algorithms to determine path existence between vertices in a directed graph is particularly important due to the important role of these algorithms in many application domains, including deductive databases, rule bases verification, and graph theory to name just a few [1-7]. These algorithms are very crucial to answer certain queries about the path existence between any two graph vertices.

Shmueli (1993) presented an algorithm that keeps two pieces of information *v.round* and *v.num* with each vertex *v* maintaining a property called conservatism in a graph *G*. Using these pieces of information and this property, the algorithm answers some negative path existence queries without performing DFS. The algorithm has the overhead of assigning these values and keeps the conservatism property in each call to DFS. The traditional approach for answering path existence queries was to start with the first vertex and explore the graph to determine if the second vertex can be reached. Then, if the second vertex can't be reached, start with the second vertex and explore the graph to determine if the first vertex can be reached. However, our algorithm solves this problem in a more efficient way as will be explained in later sections.

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 [7], and in answering the generalized forms of the same generation query in deductive databases [2,4]. In this matrix, the rows represent some paths in the graph starting from the roots/source vertex to the leaves. Basically, depth-first search is used to create the paths of the graph. Instead of storing every vertex 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 vertex in the matrix, a different technique is used; for the first visit to the vertex, it is entered into the matrix and the coordinates of its location is recorded. On subsequent visits, instead of entering the vertex itself, its coordinates are entered into the matrix (a pointer to the already stored vertex). In this way, only a single copy of each of the graph's vertex is guaranteed to be entered in the matrix. Moreover, there will be only one entry (either a vertex or a pointer) in the matrix for each edge in the graph. In Figure 1 (b), the matrix representation of the graph given in Figure 1 (a) 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 vertex *s* and *n*). An important advantage of this matrix structure is that it stores a path from each root to each vertex that is reachable from that root. 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 2. 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. The 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 vertex is sufficient. This is due to the fact that we are interested in the path lengths and not in the stored vertex themselves, from the matrix structure.



	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) GRAPH FORM (b) MATRIX REPRESENTATION

FIGURE 1. DIRECTED GRAPH IN (a) GRAPH FORM (b) MATRIX REPRESENTATION

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

6,6	7,6	8,0	8,1	9,1	9,2	10,1	10,2	11,2	11,3
a	b	n	5,2	K	6,3	l	9,1	J	6,3

FIGURE 2. THE MATRIX AS LINEAR ARRAY

Row No.	Row Beginning	Row End	Row End Entry
0	0	2	o
1	2	3	<0,2>
2	3	6	d
3	5	5	<2,6>
4	1	2	<1,2>
5	2	3	<2,3>
6	3	6	a
7	6	6	b
8	0	1	<5,2>
9	1	2	<6,3>
10	1	2	<9,1>
11	2	3	<6,3>

FIGURE 3. (a) MATRIX REPRESENTATION ROW BEGINNINGS AND ENDINGS

Vertex No.	Vertex	Vertex Coordinates <crd_1,crd_2>	Vertex Timestamps[d,f]
1	s	<0,0>	[1,30]
2	r	<0,1>	[2,15]
3	o	<0,2>	[3,4]
4	p	<1,2>	[5,14]
5	g	<2,3>	[6,13]
6	f	<2,4>	[7,12]
7	e	<2,5>	[8,11]
8	d	<2,6>	[9,10]
9	q	<4,1>	[16,29]
10	m	<5,2>	[17,28]
11	i	<6,3>	[18,27]
12	h	<6,4>	[19,26]
13	c	<6,5>	[20,25]
14	a	<6,6>	[21,22]
15	b	<7,6>	[23,24]
16	n	<8,0>	[31,38]
17	k	<9,1>	[32,33]
18	l	<10,1>	[34,37]
19	j	<11,2>	[35,36]

FIGURE 3. (b) DIRECT ACCESS STRUCTURE OF THE VERTICES IN THE MATRIX

3. THE PATH EXISTENCE QUERY ALGORITHM

The algorithm is implemented by the procedure Path_Existence_Query as shown in Figure 4. After determining the vertices pair (v_1,v_2), the algorithm proceeds to determine whether there is a path between these vertices. The path can be from v_1 to v_2 or from v_2 to v_1. The procedure uses the coordinates of the vertices in the matrix and their timestamps [*d*,*f*], where *d*[*v*] records when *v* is first discovered, and *f*[*v*] records when the search finishes examining *v*'s adjacency list, determined by a standard DFS as shown in Figure 3. (a) and (b), to determine the path existence. If both vertices belong to the same source vertex, then timestamp intervals may enable us to determine whether v_1 is a descendent of v_2 or vice versa. If v_1's timestamp interval is contained entirely within v_2's timestamp interval then v_1 is a descendent of v_2 and vice versa. Otherwise, the procedure calls the function Directed_Path_Existence_Query (Figure 5.) to determine the path existence in the other direction (i.e., from v_1, whose coordinates are crd_{11} and crd_{12}, to v_2, whose coordinates are crd_{21} and crd_{22}). If a path exists, then the path existence query returns TRUE. Otherwise, the query returns FALSE. The algorithm is invoked with the vertices pair as the input, and then uses the matrix representation, the direct accessible structure of the matrix vertices (including their timestamps intervals, and the row beginnings and ends) to check whether there is a path between the two vertices or not.

Example: For the matrix representation in Figure 1 (b) and its associated structures in Figure 3 (a) and (b), to check if there is a path from q to h, the algorithm uses the direct accessible structure to find the coordinates of the vertices in the matrix (coordinates of q: <4,1> and coordinates of h: <6,4>). It also determines the timestamps intervals for the vertices (timestamp interval for q is [16,29]; and timestamp for h is [19,26]). Since $crd_{11}(q) < crd_{21}(h)$, $crd_{12}(q) < crd_{22}(h)$, and the timestamp interval of vertex h is contained entirely within the timestamp interval of q, then there is a path from q to h (using Nesting of Descendent Intervals Theorem), and there is no need to call the Directed_Path_Existence_Query function. To check if there is a path from q to g, the algorithm uses the direct accessible structure to find the coordinates of the vertices in the matrix (coordinates of q: <4,1> and coordinates of g: <2,3>). It also determines the timestamps intervals for the vertices (timestamp interval for q is [16,29]; and timestamp for g is [6,13]). Since $crd_{11}(q) > crd_{21}(g)$, and the timestamp interval of vertex g is not contained entirely within the timestamp interval of q, then the algorithm calls the function Directed_Path_Existence_Query(4,1,6,4) and starts with q and jumps to the row end of row 4 where it finds the pointer <1,2>, then follows the pointer to row 1 where it finds the pointer <0,2> at the row end, where it finds that the pointer is beyond the vertex g, and thus it prunes the search at <0,2> and moves to the second path (the next row in the matrix) where it finds the vertex g. Thus, there is a path between q and g.

There are three important points to notice:

(1) The algorithm uses the coordinates of the vertices, the timestamp intervals associated with each vertex, and the Nesting of Descendent Intervals theorem to eliminate one of the searches that are

always used to determine whether there is a path between two vertices v_1 and v_2. It also answers some path existence queries using the timestamps intervals. On the other hand, to determine if there is a path between v_1 and v_2, DFS-based approaches calls DFS algorithm twice: DFS(v_1,v_2) and DFS(v_2,v_1).

(2) The algorithm uses the row beginnings and endings of the matrix (it does not check all the vertices on the paths). If the graph has long paths among vertices, then the algorithm will skip most of these vertices. On the other hand, DFS-based approaches consider all vertices along the paths.

(3) The algorithm uses the properties of the matrix representation, to prune or bound the traversing of the vertices in the matrix, since it terminates the search once a pointer goes beyond the vertex that the algorithm tries to reach and then backtracks from there. On the other hand, DFS-based approaches may explore irrelevant parts of the graph. If DFS starts with v_1, then it will consider all vertices reachable from that vertex until it reaches v_2 (if v_2 is reachable).

Notice that the first case, which says that if one vertex lies before another vertex in the matrix and the timestamp interval of the second vertex is contained entirely in the timestamp interval of the first vertex, then there is a path from the first vertex to the second vertex, is always true. If there is such a path, then the second vertex should be visited starting from the first vertex. The second vertex should not have been visited before, for if this is the case, then it cannot come after the first vertex in the matrix structure. These properties can be formalized in the following:

(a) Path_Existence_Query algorithm needs one call to a Directed_Path_Existence_Query algorithm. The algorithm uses the coordinates of the vertices, the timestamp intervals associated with each vertex, and the Nesting of Descendent Intervals theorem to eliminate one of the searches that are always used to determine whether there is a path between two vertices v_1 and v_2. It also answers some path existence queries using the timestamps intervals. On the other hand, to determine if there is a path between v_1 and v_2, DFS-based approaches call DFS algorithm twice: DFS(v_1,v_2) and DFS(v_2,v_1).

(b) Path_Existence_Query algorithm uses the row beginnings and row endings instead of visiting all vertices in the graph. The algorithm uses the row endings of the matrix (it does not check all the vertices on the paths). If the graph has long paths among vertices, then the algorithm will skip most of these vertices. On the other hand, DFS-based approaches consider all vertices along the paths.

(c) Path_Existence_Query algorithm prunes the graph in searching. The algorithm uses the properties of the matrix representation to prune or bound the traversing of the vertices in the matrix, since it terminates the search once a pointer goes beyond the vertex that the algorithm tries to reach and then backtracks from there. On the other hand, DFS-based approaches may explore irrelevant parts of the graph. If DFS starts with v_1, then it will consider all vertices reachable from that vertex until it reaches v_2 (if v_2 is reachable).

As mentioned before, an algorithm that keeps two pieces of information *v.round* and *v.num* with each vertex *v* maintaining a property called conservatism in a graph *G* was presented by [6]. Using these pieces of information and this property, the algorithm answers some negative path existence queries without performing DFS. The algorithm has the overhead of assigning these values and keeps the conservatism property in each call to DFS. The Path_Existence_Query Algorithm, on the other hand, answers many positive path existence queries using vertices timestamps, which are computed in a preprocessing step. In addition, the algorithm reduces the number of vertices that need to be visited to answer a path existence query.

4. PERFORMANCE EVALUATION OF THE PATH EXISTENCE QUERY ALGORITHM

To show that The Path Existence Query Algorithm outperforms other competitive algorithms, a simulation study to compare our algorithm with the algorithm presented in [6], which is considered to be one of the best algorithms, has been performed. A set of graphs, each with 5000 edges, with different outdegrees has been used in answering one hundred path existence queries and the amount of time needed was computed. These results are shown in Figure 6. The results show that our algorithm outperforms the algorithm presented in [6] due to the fact that: First, the number of nodes visited in The Path Existence Query Algorithm (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 The Path Existence Query Algorithm. This is due to the fact that larger outdegree values of the underlying graph generate longer paths of the graph, which results in skipping larger number of nodes in the graph. The Path Existence Query Algorithm considers only the relevant part of the graph by examining the set of nodes that are connected to the selected node from which the search to start and involves less computation than the Shmueli's algorithm. The Path Existence Query Algorithm answers most of the path existence queries in a very efficient way.

```
Procedure Path_Existence_Query(vertex1,vertex2)
{
        (crd11, crd12 )= Coordinates of vertex1 in Matrix;
        (crd21 , crd22 )= Coordinates of vertex2 in Matrix

        if(crd11<crd12) ||(crd11=crd12 && crd12<crd22) then {
                // Vertex1 comes before vertex2 in Matrix
                if(d[vertex1] <= d[vertex2] && f[vertex1] >= f[vertex2])  then
                        print There is a path from vertex1 to vertex2 (in G);
                else {
                        current_vertex=vertex2;
                        if(Directed_Path_Existence_Query(crd21,crd22,crd11,crd12)==1) then
                                print There is a path from vertex2 to vertex1 (in G);
                }
        }
        else {
                // Vertex2 comes before vertex1 in Matrix
                if(d[vertex2] <= d[vertex1] &&  f[vertex2] >=f[vertex1])  then
                        print There is a path from vertex2 to vertex1 (in G);
                else {
                        current_vertex=vertex1;
                        if(Directed_Path_Existence_Query(crd11,crd12,crd21,crd22)==1) then
                                print There is a path from vertex1 to vertex2 (in G);
                }
        }
}
```

FIGURE 4. PATH_EXISTENCE_QUERY ALGORITHM

```
Function Directed_Path_Existence_Query(crd11,crd12,crd21,crd22)
{
  Direct_Access_Structure [current_vertex].color=1;  //GRAY
  j=crd11;

  do {
        if(j <=crd21 || the last entry in the row is not a pointer) then {
                current_vertex= last vertex in the current row;
                if(j == crd21 && crd12 <= crd22)
                        return 1 and exit;
        }
        else {
                current_vertex= vertex whose coordinates are at row end;
                if(Direct_Access_Structure [current_vertex].color==0) then
                   return (Directed_Path_Existence_Query(coordinates of current_vertex, crd21,crd22));
        }
        j=j+1;
  } while(Row_Beg[j] > crd12 && Direct_Access_Structure [current_vertex].color==0);

  Direct_Access_Structure[current_vertex].color=2;  // 2: BLACK
}
```

FIGURE 5. DIRECTED_PATH_EXISTENCE_QUERY ALGORITHM

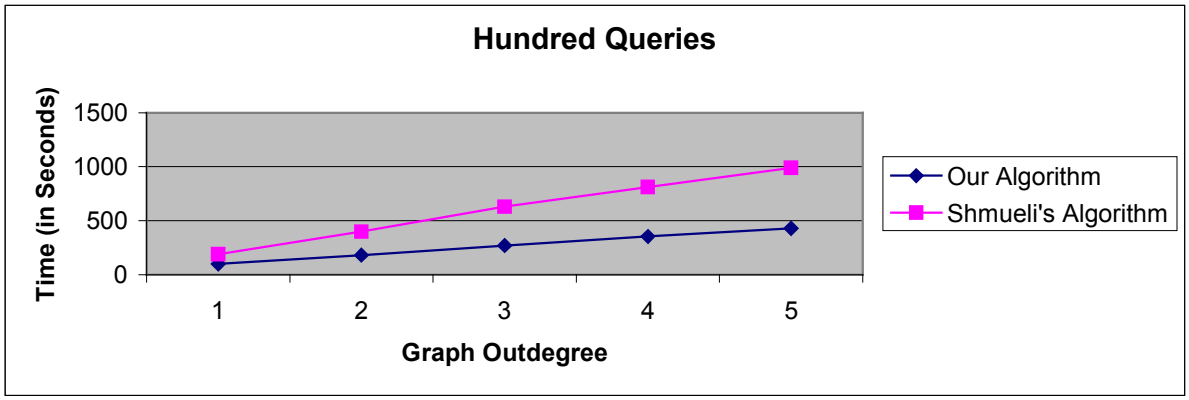


FIGURE 6. PATH_EXISTENCE_QUERY ALGORITHM VS. SHMUELI'S ALGORITHM

5. CONCLUSION

This paper presents an efficient algorithm to check path existence between vertices in a directed graph. The algorithm utilizes a special matrix representation of the graph to eliminate one of the searches that are always performed to answer such queries and prunes the search of irrelevant parts of the graph using the structures associated with the rich matrix representation. Finally, a simulation study demonstrates that our algorithm outperforms other approaches for answering the path existence query.

REFERENCES

[1] Arman, N., "Graph Representation Comparative Study," Proceedings of the 2005 International Conference on Foundations of Computer Science (FCS'05), June 27-30, 2005, Las Vegas, USA, Accepted.

[2] Arman, N., "An Efficient Algorithm for the Generalized Partially Instantiated Same Generation Query in Deductive Databases," The International Arab Journal of Information Technology, pp. 142-146, Vol. 1, No. 1, 2004.

[3] Arman, N., Richards, D., and Rine, D., "Structural and Syntactic Fault Correction Algorithms in Rule-Based System," International Journal of Computing and Information Sciences, pp. 1-12, Vol. 2, No. 1, 2004.

[4] Arman, N., " An Intelligent Algorithm for the Generalized Fully Instantiated Same Generation Query in Deductive Databases," Proceedings of the 4th International Arab Conference on Information Technology (ACIT'2003), pp. 224-228, December 20-23, 2003, Arab Academy for Science and Technology, Alexandria, Egypt.

[5] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001), *Introduction to Algorithms*, McGraw-Hill Book Company.

[6] Shmueli, O., (1993), Dynamic Cycle Detection, *Information Processing Letters*, Vol. 17, 185-188.

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