한국산업정보학회논문지 제5권 제1호 (2000.3)

Efficient Evaluation of Path Algebra Expressions

Tae-kyong Lee*

Abstract In this paper, an efficient system for finding answers to a given path algebra expression in a directed acylic graph is discussed, more particulary, in a multimedia presentration graph. Path algebra expressions are formulated using revised versions of operators next and until of temporal logic, and the connected operator. To evaluate queries with path algebra expressions, the node code system is proposed. In the node code system, the nodes of a presentation graph are assigned binary codes (node codes) that are used to represent nodes and paths in a presentation graph. Using node codes makes it easy to find parent-child, predecessor-sucessor relationships between nodes. A pair of node codes for connected nodes uniquely identifies a path, and allows efficient set-at-a-time evaluations of path algebra expressions. In this paper, the node code representation of nodes and paths in multimedia presentation graphs are provided. The efficient algorithms for the evaluationof queries with path algebra expressions are also provided.

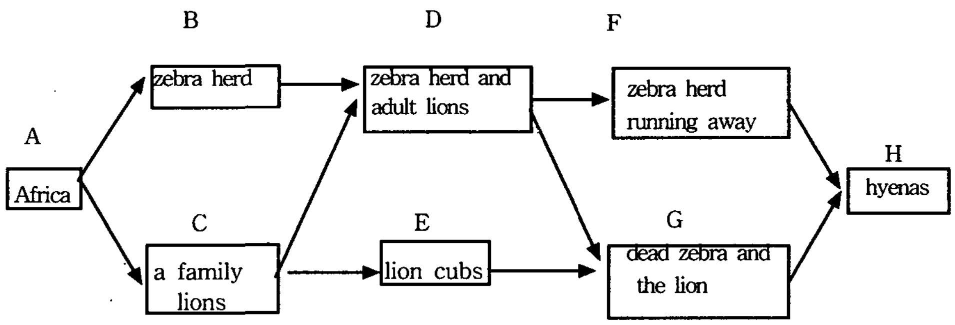
1. Introduction

In this paper, an efficient system for finding answers to a given path algebra expression in a directed acylic graph is discussed. Queries specify path expressions using operators next, until and connected. Among these operators, the semantics of next and until are similar to the semantics of next and until opertors of temporal logic[5]. However, the operators used in the queries are defined on paths of a directed acyclic graph. In a similar way, the connected operator has semantics similar to the operator eventually of temporal logic. Operators next, until and connected allow us to find and extract paths in a directed acyclic graph.

Directed acyclic graphs are used to model data in many applications. The approach using multimedia databases as a general application is presented here where directed acyclic graphs are naturally used as objects in the database. In the fields of digital libraries, electronic classroom and distance learning, the tasks of creating, viewing, querying and manipulating multimedia presentations from databases will be very common. Figure 1.1 shows a presentation which is in the form of a directed acyclic graph (DAG) with nodes representing multimedia strreams. A stream can be of type video, audio, text, or image. A stream of type video consists of a sequence of frames, where each frame contains content objects and relationships among content objects. Figure 1.2 gives an object-oriented data model for presentation graphs, streams, frames and content objects.

Example 1.1; The multimedia presentation in Figure 1.1 depicts a hunting scene of a zebra by lions. After the Africa (node A), a zebra herd (node B) and a family of lions (node C) are observed as parallel streams. The herd and lions are seen at the same time and the adult lions approach the herd for the hunt (node D). The cubs of the lions stay behind (node E). The adult lions hunt down a zebra and their cubs join the adults for a feast (node G). The rest of the zebra herd

* School of information design at ulsan University



< Figure 1.1> A multimedia presentation

```
class Pres_Graph [
        name: String;
        other attributes;
        Nodes: {Pres_Node};
        Edges: {Pres_Edge}];

  class Pres_Node [
        name: String;
        type: String;
        rep_frame: <Frame>;
        other attributes];

  class Pres_Edge
        [<Pres_Node>];

* [ ], {}, <> denote tuple, set and
  sequence objects, respectively
```

```
class Stream: inherits
                from Pres_Node[
     graph-in: Pres_Graph;
     child-nodes; {Pres_Node};
      objects; {C_Object};
   other attributes];

  class Frame[
      name: String;
      objects: {C_Object};
      other attributes];

 class C_Object [
      name: String;
      frame-in: Frame;
      other attributes];
```

< Figure 1.2> A data model for multimedia presentation graphs

flee (node F). Finally
hyenas are shown approaching the secen (node H).

For querying purposes, assertions on the nodes of presentation graphs can be made based on the content objects they contain. These assertions are specified in the form of predicates. For example, node D satisfies the following predicates:

- node D contains the content object zebra herd,
- node D contains the content object adult lions.

In this paper, GCalculus/S[10] is used to illustrate queries on presentation graphs (a detailed presentation of GCalculus/S is given in [10]).

Example 1.2 The query "Give all paths where a video stream with a zebra object is immediately followed by a video stream with a lion object" can be expressed in GCalculus as

$\{x|\ (\exists g)(\text{Pres\_Graph}(g)\ \wedge$

$\quad(\exists s1, \exists s2)(\text{Stream}(s1)\ \wedge\ \text{Stream}(s2)\ \wedge\ \{s1,s2\} \subseteq g.\text{Nodes}\ \wedge$

$\quad<<g,x,s1>>[[s1]](\exists o1)(\text{C\_Object}(o1)\ \wedge o1.\text{name} = \text{"zebra"} \wedge\ o1 \in s1.\text{frame-in.objects})$

$\quad\quad \mathbf{X}{:}x\ \ [[s2]](\exists o2)(\text{C\_Object}(o2)\ \wedge\ o2.\text{name} = \text{"lion"}\ \wedge\ o2 \in s2.\text{frame-in.objects})))\}$

This query asks for paths (of length 1) that satisfy the formula p **X**(next) **q**, where p evaluates to true for streams with zebra objects and q evaluates to true for streams with lion objects. The paths that satisfy the query for the presentaton graph in Figure 1.1 are <B D> and <D G>,

The immediate precedence relationship between two nodes of a DAG is captured by the **X**(next) operator. The next operator is a binary operator (unlike the next operator of temporal logic which is a unary operator). The **p X q** relation holds for the path <v1 v2> if

- **p** evaluates to true to v1.
- **q** evaluates to true to v2.

Example 1.3 The query "Give all paths between a video stream with a zebra object and a video stream with a lion object" can be expressed in GCalculus/S as

$\{x|\ (\exists g)(\text{Pres\_Graph}(g)\ \wedge$

$\quad(\exists s1, \exists s2)(\text{Stream}(s1)\ \wedge\ \text{Stream}(s2)\ \wedge\ \{s1,s2\} \subseteq g.\text{Nodes}\ \wedge$

$\quad<<g,x,s1>>[[s1]](\exists o1)(\text{C\_Object}(o1)\ \wedge o1.\text{name} = \text{"zebra"} \wedge\ o1 \in s1.\text{frame-in.objects})$

$\quad\quad C{:}x\ \ [[s2]](\exists o2)(\text{C\_Object}(o2)\ \wedge\ o2.\text{name} = \text{"lion"}\ \wedge\ o2 \in s2.\text{frame-in.objects})))\}$

This query asks for paths that satisfy the formula **p** C(connected) **q**, where p evaluates to true for stream with zebra object and q evaluaes to true for streams with lion objects. The paths that satisfy the query for the presentation graph in Figure 1.1 are <B D>, <B G>, and <D G>.

The operator **C**(connected) is used to assert the existence of a path between two nodes in a graph. The **C** operator is a binary operator. **p C q** relation holds for a path <v1 ... vn> if

- **p** evaluates to true for v1,
- **q** evaluates to true for vn.

Example 1.4 The query "Give all paths where the zebra object appears in all consecutive streams until a stream with a lion object" is expressed in GCalculus/S as

$\{x|\ (\exists g)(\text{Pres\_Graph}(g)\ \wedge$

$\quad(\exists s1, \exists s2)(\text{Stream}(s1)\ \wedge\ \text{Stream}(s2)\ \wedge\ \{s1,s2\} \subseteq g.\text{Nodes}\ \wedge$

$\quad<<g,x,s1>>[[s1]](\exists o1)(\text{C\_Object}(o1)\ \wedge o1.\text{name} = \text{"zebra"} \wedge\ o1 \in s1.\text{frame-in.objects})$

$\quad\quad U{:}x\ \ [[s2]](\exists o2)(\text{C\_Object}(o2)\ \wedge\ o2.\text{name} = \text{"lion"}\ \wedge\ o2 \in s2.\text{frame-in.objects})))\}$

This query asks for paths that satisfy the formula **p** U(until) **q**, where **p** holds true for the streams with a zebra object and q holds true for streams with a lion object. For the multimedia presentation graph in Figure 1.1, the resulting paths for the query are <D G> and <B D G>.

The semantics of repetition with a terminating condition is captured by the **U**(until) operator which is also a binary operator. p **U** q holds for a path <v1 ... vn-1 vn> if

- p evaluates to true for all v1, ... , vn-1.
- q evaluates to true for vn.

In this paper, the algorithms to efficiently implement the path algebra operators **X, C,** and **U** are given. Traditional graph algorithms for checking adjacency or connectivity of every pair of nodes in a graph can be very time consuming when used in implementing path algebra operators. The approach in this paper is to use an encoding system, called the node code system. The node code system assigns binary code to nodes of a presentation graph, and the node codes are used to efficiently evaluate path algebra operators. This approach eliminates the need to traverse presentation graphs.

In section 2, the related work are presented briefly. Section 3 describes the generation of node codes, presents their properties, and defines two different orders; namely, alphabetical ordering and level-first ordering on node codes. Section 4 gives efficient evaluations for the path algebra operators **X**, **C**, and U. Section 5 describes the approach to the implementation of the query processing techniques discussed in section 4. Section 6 is the conclusion.

2. Related Work

Graphs in databases have been used by different researchers in different contexts. In the database query and visualization systems G+[3,4] and Hy+[1,2], the database is visualized as a graph, and a query is formulated as a set of graphs which are viewed as patterns to be matched against the database. In the Graph-Oriented Object Database model[6,7,8], the emphasis is on representing the database scheme and the database instance as a graph. Database queries are expressed using five primitive graph operations (node and edge additions and deletions, and a duplicate eliminator). There is also a graphical user interface for formulating queries and visualization of results[6]. In this model, objects are represented as nodes in a graph, and relationships and properties of objects are represented as edges. Another alternative, which is used in many object-oriented data models, is to represent relationships of objects as object-valued attributes, and to use "path expressions" with an SQL-like language, such as OQL. However, for applications where graphs are needed to model data, supporting graphs explicitly in the data model and in the query language (in addition to supporting other bulk types such as lists and sets) is very beneficial. GraphDB is a good example which explicitly allows modeling and querying of graphs[9]. In GraphDB, there are three kinds of object classes, namely, simple classes, link classes, and path classes. A link class has references to source and target simple objects (in addition to having other components). Path objects are defined using regular expressions of link objects. GraphDB provides an SQL-like query language with multiple levels of queries and function calls.

Mostly, the research on graphs in databases has concentrated on modeling graphs and query languages on graphs. Although these query languages have the expressive power for formulating complex queries, the evaluation of such queries are quite expensive due to graph traversals that are required during evaluation. I am not aware of any work that takes a systematic approach to eliminate explicit graph traversal during the evaluation of queries with graphs.

In the context of multimedia databases, temporal operators such as next, until, eventually have been used for expressing queries on multimedia streams. In[12] and [11], these temporal operators are used in a language called Hierarchical Temporal Logic (HTL), for similarity based retrieval of video segments. The video streams are considered to consist of hierarchically structured video segments, and the temporal operators can be used to specify properties of sequencies of video segments at any level. However, although video segment sequences can be queried at different levels, the segments in the same level appear to be in a single continum, that is, without any branching. This is not the case when paths in directed acyclic graphs, that can only be modeled with branching time[10] as in Computational Tree Logic[5], are considered.

3. Physical Data Model

Now, a scheme, called the node code system, that is used for the evaluation of operators and for the identification of paths in a DAG with a single source node (a node with no incoming edges). These codes are in the form of binary strings. Although every node code uniquely identifies a node, a node may have more than one node codes, if that node is connected from the source node by multiple different paths. Given all the node codes of two nodes, it is easy to check if the two nodes satisfy the parent-child relationship (next), or if they are connected (connected) via simple comparisons. If there exists a path between two nodes, it is possible to create the paths between these two nodes using their node codes without traversing the nodes of the actual path.

3.1 Assigning Node Codes

A DAG, that has a single source node, is assumed. If it has more than one source nodes, it is possible to convert the DAG into a DAG with one source node by simple adding an extra node into the graph and making a connection (by an edge) from the new node to every source node of the DAG.

The folowing procedure is used to assign node codes to the nodes of a DAG. Node codes are denoted as a sequence of binary digits (binary strings) such as <a1 a2 ... am> where each ai (i=1...m) is either 0 or 1. Binary string, X* indicates 0 or more repetition of X, where X is either 0 or 1, or a substring given in parenthesis. Similarly, X+ indicates one or more repetition of X and Xi indicates exactly i repetition of X. The source node is assigned the node code 1. The initial call to the algorithm is Assign_Node_Code(source, 1).

Algorithm: Assign_Node_Code(v,<a1 a2 ... ax>) where ai = 0 or 1, for all i=1..x.
Input: A node v, a node code <a1 a2 ... ax> of node v.
Output: Children of node v are assigned new node codes.

```
1  c=number of children of v (C1 ... Cc are the children of v)
2  for i=1 to c do
3  begin
4      Add node code <a1 ... ax 0 1i-1> to the list of node codes for Ci
5      Assign_Node_Code(Ci, <a1 ... ax 0 1i-1>)
6  end
```

In Figure 3.1, an ordered tree, that shows the nodes of the DAG in Figure 1.1 and the node codes assigned to them, is seen. A depth-first traversal of the tree in Figure 3.1 gives the order of nodes visited by the algorithm Assign_Node_Code. Note that some nodes are visited more than once, and are assigned more than one node codes. In fact, the number of node codes for a node v is equal to the number of paths from the source node to v. For example, there are two different paths from the source node A to node D (<A B D> and <A C D>) and therefore there are two node codes (100, 1010) assigned to node D (See Figure 3.2). The source node has only one node code (1).

In the following lemmas, some useful properties of node codes are illustrated and highlighted.

Lemma 3.1: Let v be a node with node code <a1 ... ax> and v0, ... vn be the children of v. Then the node code of vl=<a1 ... ax 0 1i>.

The proof of Lemma 3.1 is obvious from line 4 of the Assign_Node_Code algorithm.

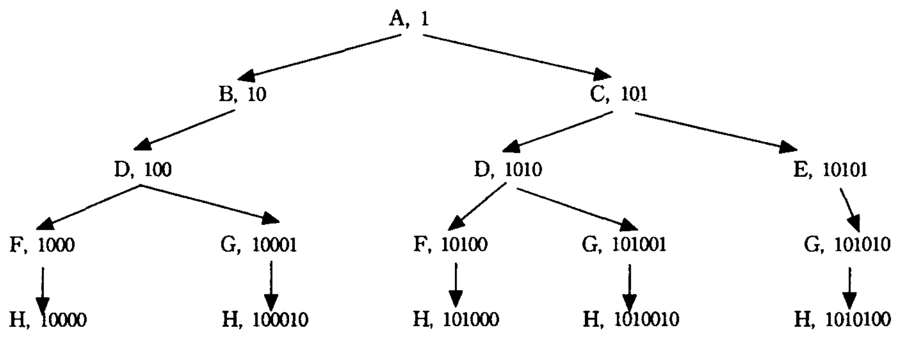
Corollary 3.1: If a node v has the node code <a1 ... ax 0 1*> then its parent has the node code <a1 ... ax>.

Lemma 3.2: All children of a node are assigned different node codes.

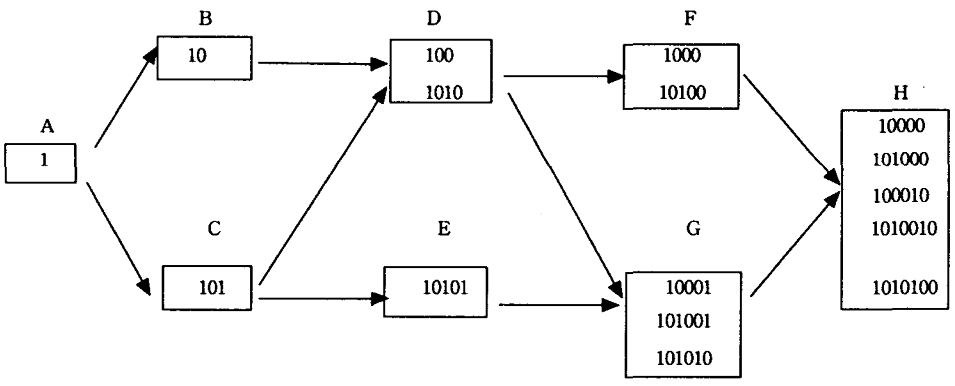
Example 3.1: Node C has the node code 101. Children of node C are nodes D and E. Node codes for the children of node C have the prefix 101 and the form 10101*. Node D has the node code 1010 which is equivalent to 101010. Node E has the node code 10101 which is equivalent to 101011.

Theorem 3.1: A node code can be assigned to only one node in a directed acyclic graph.

Proof: This lemma is proven by induction on the length of node codes.



<Figure 3.1>. Node code assignment for the nodes of DAG in Figure 1.1



<Figure 3.2> DAG of Figure 1.1. with the assigned node codes

hypothesis: All node codes of length1 are assigned to different nodes. This is obviously true as there is only one node code of length 1 and it is assigned to the source node.
induction step: Let us assume that all node codes of length < k are assigned to different nodes. Let us further assume that two nodes, v1 and v2, are assigned the same node code of length k. Their parents, pv1 and pv2, can not be the same as no children of a node are assigned the same node code by Lemma 3.2. Therefore, their parents pv1 and pv2 should be different nodes, and their node codes should be the same due to Corollary 3.1. However, the node codes of parents pv1 and pv2 should have a length less than k, which leads us to a contradiction as we assumed that node codes of length < k are assigned to different nodes in our induction step.

It is easy to see that node codes are binary strings in the form 1(01*)*. The source node has node code 1 and line 4 of the algorithm Assign_Node_Code only adds 01*s. It should be clear at this point that if a node v1 is an ancestor of node v2, and v1 has a node code <a1 ... ax>, then v2 has node code <a1 ... ax (01*)+>.

Theorem 3.2: A node code for a node v defines a unique path from the source node to v.

Proof: Since node v is an descendent of the source node and the source node has the node code 1, then v has a node code of the form <1 A1 ... Am> where each Ai (i=1..m) is a binary string of the form (01*). This node code defines a path from the source node to v traversing the nodes with node codes <1 A1>, <1 A1 A2>, ..., <1 A1 ... Am>, in order. Since each node code is assigned to a unique node (by Theorem 3.1), this path is unique.

Corollary 3.2: Assume that two nodes, v1 and v2, have node codes <a1 ... ax>, and <a1 ... ax A1 ... Am> (where each Ai (i=1..m) is a binary string of the form (01*)), respectively. Then, v1 and v2 are connected, and these two node codes define a unique path of length m from v1 to v2.

From Corollary 3.2, it is clear that v1 is an ancestor of v2 considering their node codes. These node codes define a unique path from v1 to v2, traversing the nodes with node codes <a1 ... ax A1>, <a1 ... ax A1 A2>, ..., <a1 ... ax A1 ... Am>.

Example 3.2: The node codes 101 and 101000 of node C and H, respectively, define a path of length 3 between node C and node H in Figure 3.1. This path is <101 1010 10100 101000>, or <D D F H>.

Definition 3.1: A path instance is path between two nodes and is uniquely specified by a pair of node codes. In other words, a path instance is a 2-tuple of the form (c1, c2) where c1 is a node code for the first node of the path and c2 is a node code for the last node of the path, and c2 is of the form c1(01*)+.

If a path instance has the length 0, the first and the last nodes are the same and it can simply be denoted by a single node code.

3.2 Ordering of Node Codes

Two different orderings on node codes are defined. The first ordering is the alphabetical ordering. Alphabetical ordering corresponds to a depth-first traversal of the graph. The node codes are assigned to the nodes in alphabetical order by the recursive algorithm Assign_Node_Code of Section 3.1. One can

alphabetical ordering		level-first ordering	
1	(A)	1	(A)
10	(B)	10	(B)
100	(D)	101	(C)
1000	(F)	100	(D)
10000	(H)	1010	(D)
10001	(G)	10101	(E)
100010	(H)	1000	(F)
101	(C)	10001	(G)
1010	(D)	10100	(F)
10100	(F)	101001	(G)
101000	(H)	101010	(G)
101001	(G)	10000	(H)
1010010	(H)	100010	(H)
10101	(E)	101000	(H)
101010	(G)	1010010	(H)
1010100	(H)	1010100	(H)

<Figure 3.3> Node code orderings for the DAG in Figure 1.1.

think of the alphabetical order as an ordering of the strings generated by a language whose letters are 0 and 1, where 0 has precedence over 1. The most useful properties of alphabetical ordering are illustrated by the next two lemmas.

Lemma 3.3: If node v has a node code <a1 ... ax>, then all nodes connected from v have a node code that comes after <a1 ... ax> in alphabetical ordering.

Proof of Lemma 3.3 follows from Corollary 3.1. For a node v with a node code <a1 ... ax>, ancestors of v have node codes which are prefixes of <a1 ... ax>. In an alphabetical order, prefix of a word precedes the word.

Lemma 3.4: If node v has a node code <a1 ... ax>, then all nodes connected from v have node codes that come before <a1 ... ax 1> wrt alhabetical ordering.

Proof of Lemma 3.4 also follows from Lemma 3.1. If a node v has a node code <a1 ... ax>, then each of its successors has a node code with prefix <a1 ... ax 0>, which comes before <a1 ... ax 1> wrt alphabetical ordering. Note that a node with the node code <a1 ... ax 1> is a sibling of node v.

Alphabetical ordering is used in limiting the search for connectivity to a closed range. This range is defined by lemmas 3.3 and 3.4 for each node code. In other words, all node codes of nodes connected from node v with node code <a1 ... ax> are between <a1 ... ax> and <a1 ... ax 1> in alphabetical ordering. This property is utilized in evaluating the connected operator in path algebra expressions (see section 5).

Example 3.3: Considering the DAG in Figure 3.2, node B is an ancestor of node F. The node for node B is 10. Node F has the node code 1000 which comes after the node code 10 (Figure 3.2) and before the node code 101 wrt alphabetical ordering, as ilisted in Figure 3.3.

Another ordering that is used on the node codes is the level-first ordering. Level-first ordering corresponds to a breadth-first travrersal of the DAG. However, in level-first ordering, children are visited with respect to alphabetical ordering of their corresponding node codes. As an example, a breadth first traversal of the nodes in Figure 3.1 gives the level-first ordering of the node codes of the DAG in Figure 1.1. This is shown in Figure 3.3.

Remark 3.1: The numbe of 0's in a node code for node v is equal to the length of a path from the source node to the node v, which is identified by that node code of v.

To sort the node codes with respect to level-first ordering, first, node codes are sorted with respect to the number of 0's they contain. The sorting among the node codes with the same number of 0's is done wrt alphabetical ordering. Level-first ordering puts node codes of sibling nodes together. This ordering is utilized in the evaluation of the next operator.

Lemma 3.5: Let nodes v0 ... vn be the children of a node v. Then, node codes of v0 ... vn are clustered together wrt level-first ordering.

Proof: If v has the node code <a1 ... ax> then each of its children vi has node code <a1 ... ax 0 1i> for i=0,...,n by Lemma 3.1. Lemma 3.5 follows from the fact that level-first ordering corresponds to a breadth-first traversal of the DAG.

Example 3.4: Nodes D and E are sibling nodes, as they are the children of node C. Node codes for nodes D and E are 1010 and 10101, respectively. These node codes appear one after another in level-first ordering (Figure 3.3).

4. Path Algebra Operators

When only node and edge information is used, finding paths that satisfy a given predicate in a directed acyclic graph requires the traversal of nodes in the graph. However, graph traversal in a database introduces extra disk I/O (possibly due to multiple accesses in the traversal of the same set of nodes in different paths of the graph). This actually corresponds to a path-at-a-time evaluation of the predicates that are specified in queries. Path-at-a-time approach is also space-inefficient as it requires handling of variable size paths (node sequences) during the evaluation of queries. In the approach adopted here, the node codes for representing paths and paths along a directed acyclic graph are used, which allows for efficient set-at-a-time evaluations of queries on paths. With set-at-a-time approach, it is possible to create optimum and efficient evaluation plans, which is especially important for queries with complex predicates on paths.

The path algebra with the three operators, **X**(next), **C**(connected) and **U**(until), is introduced to specify evaluation plans for queries on directed acyclic graphs using the node code system. The operators **X**, **C**, and **U** are binary operators that take two sets of path instances as input and return a set of path instances that satisfy the semantics of the operators which were discussed in section 1.

A path algebra expression is defined as follows:

1) A predicate is a path algebra expression. In this case, the path algebra expression is satisfied by path instances of length 0 (single nodes).

2) (pexpl **X** pexp2), (pexpl **C** pexp2), (pexpl **U** pexp2) are path algebra expressions where pexpl and pexp2 are path expressions.

Given two paths instances (ab, ae) and (bb, be) satisfying path algebra expressions pexpl and pexp2, respectively, the evaluation of (pexpl **X** pexp2) returns the path instances (ab, be) if the node code bb is a child of ae (i.e., bb is of the form ae01*). Similarly, the evaluation of (pexpl **C** pexp2) returns the path instance (ab, be) if the node code bb is a sucessor of ae (bb is of the form ae(01*)+). Evaluation of the path algebra expression (pexpl **U** pexp2) returns the path instances where the path algebra expression pexpl is repeatedly satisfied one or more times before the path algebra exprssion pexp2 is satisfied.

5. Evaluation of Path Algebra Expressions Using Node Codes

In the approach here, path algebra expressions are evaluated set-at-a-time where the sets are collections of path instances. All path instances that satisfy the path algebra expression given in a query are output of the query. Using path instances and node codes, the actual traversal of the nodes along a path in the graph is avoided during the evaluation of path algebra expressions.

5.1 *X(next)*

A path algebra expression of the form (pexpl **X** pexp2) is evaluated after the evaluation of pexpl and pexp2 are done. It is assumed that all path instances that satisfy the path algebra expressions pexpl and pexp2 are found and stored in tables A and B, respectively. It is also assumed that the path instances in A are denoted as (aib, aie) (i=1..n where |A|=n), and the path instances in B are denoted by (bjb, bje) (j=1..m where |B|=m). aib and aie denote the first and the last node codes for the path instances in table A. Similarly, bjb and bje denote the first and the last node codes of the path instances in table B. When comparing two path instances PIA from table A and PIB from table B to check if they satisfy the next relation, the only thing to be done is to compare the last node code in PIA (aje) and the first node code in PIB (bjb). (pexpl **X** pexp2) evaluates to true, only if the node with the node code bje is a child of the node with the node code aie.

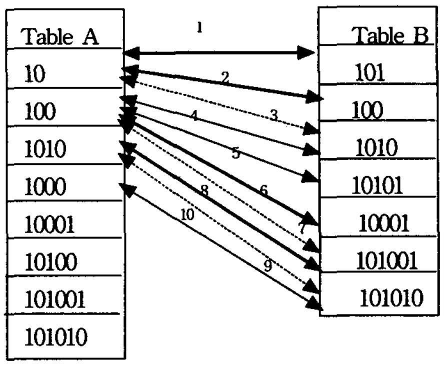
It starts with sorting tables A and B in levle-first ordering. Path instances in table A are sorted on the end node codes (aie), and the path instances in table B are sorted on the beginning node codes (bjb). Level-first ordering puts all ancestors before successors, and the same-level nodes (nodes with the same number of 0's) are ordered alphabetically.

Lemma 5.1: If v1 and v2 are two nodes with node

Table A
10
100
1010
1000
10100
101001
101001
101010

Table B
101
10101
10001
101001
101010
100

<Figure 5.1.a>. Input Tables



<Figure 5.1.b>. Sorted input tables and EvalNext

codes <a1 ... ax> and <b1 ... by>, respectively, and <a1 ... ax> precedes <b1 ... by> in level-first ordering, the set of node codes for the children of v1 (if v1 has children) precede the set of node codes for the children of v2 (if v2 has children).

Proof: If <a1 ... ax> and <b1 ... by> are in the same level (i.e., they contain the same number of 0's) and <a1 ... ax> precedes in alphabetical order, <b1 ... by>, then the node codes of children of v1 (that are derived from <b1 ... by>) precede the node codes of the children of v2 (that are derived from <b1 ... by>) in level-first ordering. If <a1 ... ax> is at a higher level (contains less 0's) than <b1 ... by>, the node codes of the children of v1 will be at a higher level than, and therefore preceding node codes of children of v2.

Using Lemma 5.1, all parent-child relationships between node codes can be dicided by a single pass over the sorted tables A and B. The following rules make it possible to compare node codes aie and bjb from tables A and B systematically.

1. If aie is the parent of bjb then the path instance (ajb, bje) is an output. Proceed with getting the next path instance in table B (b(j+1)b, b(j+1)e), and check if b(j+1)b is a child of aie.
2. If aie is not a parent of bjb, then there are two possibilities,
   a) bjb precedes the first child of aie (bjb < aie0) wrt level-first ordering. Then all children of aie will appear after bjb, and, read the next path instance in table B and check it with (aib, aie).
   b) bjb succeeds all children of aie (i.e., comes after aie01* wrt level-first sorting), in which case it can be concluded that there are no more children of aie in table B. So, read the next path instance in table A (a(i+1)b, a(i+1)e), and check if a(i+1)e satisfies the next relationship with bjb.

Lemma 5.2: Rules 1 and 2 are sound and complete in the sense that all path instances that satisfy a given path algebra expression (pexp1 ***X*** pexp2) are output and no other instances are output.

Proof: The soundness of the above rules is trivial from rule 1 since the path instances are produced as outputs if the next relationship holds true. To show the completeness for the above rules, it has to be shown that the rules do not miss any possible next relationship for aie. By Lemma 3.5, it is known that all children of aie are clustered together. Since the first child of aie has the smallest node code (wrt level-first ordering) among children, aie is not compared to node codes that precede aie0 (rule 2.a). Similarly, it is not necessary to compare node codes that succeed the largest child of aie as the child node codes are clustered wrt level-first ordering (rule 2.b). Finally, it has to be shown that the parent of bjb is not missed. Path instances in table A are discarded only by rule 2.b. Since path instances, which have children with node codes smaller than bjb wrt level-first ordering, the parent of bjb is not missed. Hence, the rules above are complete.

Example 5.1: Let us give an example to demonstrate the evaluation of path algebra expressions of the form (pexpl X pexp2). The query of Example 1.2 asks for path instances that satisfy p X q, where p evaluates to true for streams with a zebra object and q evaluates to true for streams with a lion object. From Figure 1.1, it is shown that p is satisifed by the nodes B, D, G, F and q is satisifed by the nodes C, E, D, G. In this case, since pexpl and pexp2 are simple predicates, path instances of length 0, which are denoted by single node codes, are used. Tables A and B that are input to the rules, are as shown in Figure 5.1.a. Figure 5.1.b shows the tables A and B after the node codes are sorted using level-first ordering.

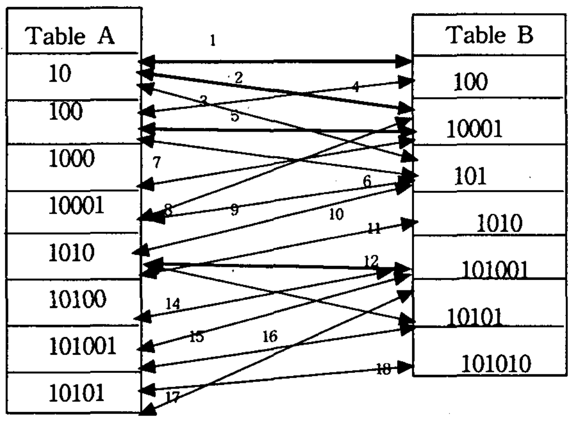
The comparison between the node codes in two tables are shown by two headed arrows in Figure 5.1.b. The order of the comparisons are indicated by numbers on the lines. A thicker line represents a next relationship between two node codes. A solid thin line indicates a condition matching rule 2.a. and a dashed thin line indicates a condition matching rule 2.b. First, node codes 10 and 101 are compared. Since the node code 101 is less than the node code 100 (first child of node code 10) wrt level-first ordering, we progress to the next node code in table B. The second comparison is between 10 and 100. Since there is a next relationship between 10 and 100, path instance (10, 100) is added to the output. The third comparison is between node codes 10 and 1010. 1010 is larger than 1001* (any child of node code 10) wrt level-first ordering. Hence, we proceed with the next node code in Table A. This continues until all path instances in either table are exhausted. As a result, the path instances (10, 100), (100, 10001), and (1010, 101001) are output.

EvalNext(A,B): Algorithm to Evaluate the ***X***(*next*) operator

input: a table A that keeps all path instances that satisfy pexpl (|A| = n)
a table B that keeps all path instances that satisfy pexp2 (|B| = m)

output: all path instances that satisfy the path algebra expression (pexpl ***X*** pexp2)

```
1 Sort the path instances in table A using level-first ordering on aie (the last node code in the path instance) for all i=1..n
2 Sort the path instances in table B using level-first ordering on bjb (the first node code in the path instance) for all i=1..m
3 i:=1, j:=1;
4 Read the entry (aib, aie) from table A
5 Read the entry (bjb, bje) from table B
6 while (TRUE) do
7 begin
8    if ($b_{jb}$ is next to $a_{ie}$) then /* $b_{jb}$ is of the form $a_{ie}01^*$ */
9    begin
10       output ($a_{ib}$, $b_{je}$) as it satisfies (pexpl X pexp2);
11       j:=j+1;
12       if (j $\leq$m) then Read the entry ($b_{jb}$, $b_{je}$) from table B
13       else return /* no more path instances in table B */
14   end
15   elsief ($b_{jb}$ is not next to $a_{ie}$ and $a_{ie}0>b_{jb}$ wrt level-first ordering) then
16   begin
17       j:=j+1;
18       if (j $\leq$m) then Read the entry ($b_{jb}$, $b_{je}$) from table B
19       else return /* no more path instances in table B */
20   end
21   elsief ($b_{jb}$ is not next to $a_{ie}$ and $a_{ie}0<b_{jb}$ wrt to level-first ordering) then
22   begin
23       i:=i+1;
24       if (i $\leq$n) then Read the entry ($a_{ib}$, $a_{ie}$) from table A
25       else return /* no more path instances in table A */
26   end
```



<Figure 5.2> Sorted input tables and the EvalConnected Algorithm

27 **end**

The EvalNext algorithm takes O(m logm + n logn) time to find out the path insances that satisfy the path algebra expression (pexpl X pexp2) once the evaluation pexpl and pexp2 are done. Sorting of the tables is the dominant step in the algorithm. Once the tables are sorted, finding the path instances that satisfy (pexpl X pexp2) takes O(m+n) time as each path instance in any of the tables A and B is read once.

An alternative is to use the brutal force approach that would require O(mn) time to match every path instance in table A to every path instance in table B and check if they satisfy the next relationship.

5.2 *C(connected)*

Evaluation of the connected operator is similar to that of the next operator. A path algebra expression of the form (pexpl C pexp2) can be evaluated once the evaluation of pexpl and pexp2 are done: the result is the list of all path instances that satisfy the expression.

It is again assumed that all path instances satisfying path algebra expressions pexpl and pexp2 are already found and stored in tables A and B, respectively. After that, the first step is to sort the tables in alphabetical ordering. Path instances in table A are sorted on end node codes (aie), and the path instances in table B are sorted on beginning node codes (bjb).

As the alphabetical ordering corresponds to a depth-first traversal of the DAG, node codes of the successors of a node v always come between the node code of v and the node code of the node that follows v in the same level (that is a sibling or a cousin of v). This is specified by Lemma 3.3 and Lemma 3.4. If node v has the node code <al ... ax>, then all nodes that are descendants of v have node codes of the form <al ... ax (01*)+>, al of which are less than <al ... ax 1>. This fact is made use in the rules to evaluate expressions of the form (pexpl C pexp2).

In the evaluation of the connected operator, node codes aie and bjb from tables A and B are compared using the following rules.

1. If aie is an ancestor of bjb then ouput path instance (aib, bje). Note the index j of table B as the backtracking point. Check path instances that follow (bjb, bje), until a path instance in B that is not connected from (aib, aie) is reached. Once all node codes connected from aie are found, return back to the back-tracking point (bjb, bje), and read the next path instance in table A (a(i+1)b, a(i+1)e).
2. If aie is not an ancestor of bjb, then there are two possibilities,

a) bjb precedes, or equal to aie (bjb $\leq$ aie) in alphabetical ordering. This means that we have not reached any ancestor of aie in table B yet, so we read the next path instance in table B.

b: bjb succeeds aie (bjb > aie) in alphabetical ordering. Since bjb is not connected to aie, bjb should also be greater or equal to (aie1) in alphabetical ordering, which means that there cannot be any ancestor of aie among the path instances that follow (bjb, bje) in table B. We read the next path instance in table A.

Backtracking is needed as a node code in Table B can be connected from more than one node code in Table A. For example assume that a node v1 in Table A is the grandparent of node v3 in Table B and node v2 in Table A is the parent of node v3. This suggests that, after we find the connectivity relation between nodes v1 and v3, we have to backtrack in table B to find the connectivity relation between nodes v2 and v3.

Example 5.2: Let us give an example to demonstrate the evaluation of path algebra expressions of the form (pexp1 C pexp2). Example 1.3 of section 1 asks for the path Instances that satisfy p C q, where p evaluates to true for streams with a zebra object and q evaluates to true for streams with a lion object. Predicate p is satisfied by the nodes B, D, G, F and q is satisfied by the nodes C, E, D, G. Tabels A and B that are iput to the rules are the same as in Figure 5.1.a. Tables A and B after the node codes sorted in alphabetical ordering are shown in Figure 5.2.

The comparison between the node codes in two tables are shown by two-headed arrows. The order of the comparisons are indicated by numbers on the lines. A thicker line represents a connected relationship between two node codes. A thin line indicates that the two node codes are not connected. For example, the node code 10 in table A is compared with node codes {100, 10001, 101}, two of them {100, 10001} being connected from 10.

First, node codes 10 and 100 are compared. Since the node code 10 is an ancestor of 100, the path instance (10, 100) satisfies the desired relationship. We progress to the next node code in table B. The second comparison is between 10 and 10001, and another ancestor is found. In the next comparison, we see that 101 is not a successor of 10 and it is greater than 10, so we backtrack to the first entry in table B (100) that is connected from 10, and progress to the next entry in table A (100). This continues until we exhaust all entries in table A.

EvalConnected(A,B): Algorithm to Evaluate the C(connected) operator
input: a table A that keeps all path instances that satisfy pexp1 (|A| = n)
a table B that keeps all path instances that satisfy pexp2 (|B| = m)
output: all path instances that satisfy the path algebra expression (pexp1 C pexp2)

```
1  Sort the path instances in table A using level-first ordering on aie (the last node code in the path instance) for all i=1..n
2  Sort the path instances in table B using level-first ordering on bjb (the first node code in the path instance) for all i=1..m
3  i:=1, j:=1;
4  Read the entry (aib, aie) from table A
5  Read the entry (bjb, bje) from table B
6  while (TRUE) do
7  begin
8     if ($a_{ie}$ is connected to $b_{jb}$) then /* $a_{ie}$0 is a prefix of $b_{jb}$ */
9     begin
10       k:=j;
11       while ($j \leq m$) and ($a_{ie}$ is connected to $b_{jb}$) do
12       begin
13          output ($a_{ib}$, $b_{je}$) as it satisfies (pexp1 C pexp2);
14          j:=j+1;
15          if ($j \leq m$) then Read the entry ($b_{jb}$, $b_{je}$) from table B
16       end
17       j:=k;
18       i:=i+1;
19       if ($i \leq n$) then Read the entry ($a_{ib}$, $a_{ie}$) from table A
```

```
20         else return;
21     end
22     elsief ($a_{ie}$<$b_{jb}$ wrt alphabetical ordering) then
23     begin
24         i:=i+1;
25         if (i ≤ n) then Read the entry ($a_{ib}$, $a_{ie}$) from table A
26         else return;
27     end
28     elsief ($a_{ie}$ ≥ $b_{jb}$ wrt to alphabetical ordering) then
29     begin
30         j:=j+1;
31         if (j ≤ m) then Read the entry ($b_{jb}$, $b_{je}$) from table B
32         else return;
33     end
34 end
```

In the algorithm above, the sorting of tables A and B takes O(n logn) and O(m logm), respectively. Once the path instances are sorted in alphabetical order in both tables, finding connected path instances takes O(m + n + o) where n and m are the sizes of the tables A and B, and o is the size of the output. The term o in the complexity O(m + n + o) result is due to the fact that backtracking is done only when entries that satisfy the connected relationship are found.

An alternative is to use the brutal force approach that would require O(mn) time to mach every path instance in table A to every path instance in table B and check if they satisfy the connected relationship.

5.3 *U(until)*

The until operator basically captures the semantics of repetition with a given terminating condition. The evaluation of the until operator is also done by repeated application of the next operator evaluation.

EvalUntil(A,B): Algorithm to Evaluate the *U*(*until*) operator

input: a table A that keeps all path instances that satisfy pexp1 (|A| = n)

a table B that keeps all path instances that satisfy pexp2 (|B| = m)

output: all path instances that satisfy the path algebra expression (pexp1 ***U*** pexp2)

```
1 k:=0;
2 while (Table $B_k$ is not empty) do
3 begin
4     Invoke EvalNext(A, $B_k$) to find all path instances that satisfy the path algebra expression (pexp1 X pexp2).
5     Store the resulting path instances in an auxiliary table $B_{k+1}$.
6 end
```

In the algorithm above, note that the table A is repeatedly used as an operand to the EvalNext algorithm, it should be sorted in level-first order only once, with complexity O(n logn). The table B, and B1 through Bm (where m is the maximum number of repetitions of pexp1 in the result of (pexp1 U pexp2)) should be sorted as the other operands of algorithm EvalNext. Overall complexity is O(m n logn).

6. Conclusion

The node code system is presented for efficient evaluation of queries with path algebra expression for application where the directed acyclic graphs are used to model the data. In the node code system, for each node in a directed acyclic graph, a node code is assigned that uniquely identifies a path from the source node to that node. A node may have more than one node codes assigned to it if there are more than one different paths that connect the source node to that node. If two node codes are connected, then they define a unique path between the two nodes they belong to. Such node code pairs are called as path instances, and are used in the evaluation of path algebra expressions avoiding explicit graph traversals. Without the graph travrsals, path algebra expressions can be evaluated set-at-a-time (compared to path-at-a-time) making it much more efficient. As only two node codes are used to represent any path in a directed acyclic graph, the burden of handling variable-length paths throughout the evaluation of a path algebra expression is also avoided.

Efficient algorithms for the evaluation of path algebra opeators X(next), U(until), and C(connected) are provided. Two orderings on node codes, namely alphabetical ordering and level-first ordering, are defined and used in evaluating the connected and next operators. The use of these orderings help reduce the complexity of the rules considerably from O(mn) to O(m logm + n logn) where m and n are the sizes of the input tables. In the algorithms, the sorting step becomes the dominating step.

References

[1] M. Consens, A. Mendelzon, "GraphLog: A Visual Formalism for Real-Life Recursion", ACM PODS Conference, 1990.

[2] M. Consens, A. Mendelzon, "Hy+: A Hygraph-based Query and Visualization System", ACM SIGMOD Conference, 1993.

[3] I.F. Cruz, A. Mendelzon, P.T. Wood, "A Graphical Query Language Supporting Recursion", ACM SIGMOD Conference, 1987.

[4] I.F. Cruz, A. Mendelzon, P.T. Wood, "G+: Recursiove Queries without Recursion", 2nd Int. Conference on Expert Database Systems, 1988.

[5] E. Emerson, "Temporal and Modal Logic", Handbook of Theoretical Computer Science, Chapter 16, editor: L. Jeeuwen, pp 995-1072, Elsevier, 1990.

[6] M. Gyssens, J. Van Den Bussche, J. Paradaens, D. Van Gucht, "A Graph-Oriented Object Database Model", IEEE Trans on Knowledge and Data Engineering, Aug., 1994.

[7] M. Gyssens, J. Paradaens, D. Van Gucht, "A Graph-Oriented Object Database Model", ACM PODS Conference, 1990

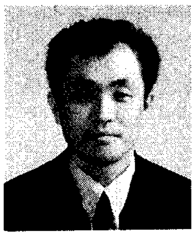
[8] M. Gyssens, J. Paradaens, D. Van Gucht, "A Graph-Oriented Object Model for Database End-User Interfaces", ACM SIGMOD Conference, 1990.

[9] R.H. Guting, "A GraphDB: Modeling and Querying Graphs in Databases", VLDB Conference, 1994.

[10] Taekyong Lee, L. Sheng, T. Bozkaya, N.H. Balkir, Z.M. Ozsoyoglu, G. Ozsoyoglu, "Querying Multimedia Presentations Based on Content", IEEE Trans on Knowledge and Data Engineering, Vol. 11, no. 3, pp 361-385, 1999.

[11] K.L. Liu, A.P. Sistla, C. Yu, N. Rishe, "Query Processing in a Video Retrieval System", IEEE Data Engineering Conference, 1998.

[12] A.P. Sistla, C. Yu, R. Venkatasubrahmanian, "Similarity Based Retrieval of Videos", IEEE Data Engineering Conference, 1997



이 태 경

1978 고려대학교 경제학과 입학,
1985 고려대학교 경제학 학사,
1988 Indiana University(Bloomington) 경제학 석사,
1998 Case Western Reserve University 컴퓨터 공학 박사 (Ph.D.),
1998.3 ~ 2000.2 경산대학교 자연과학대학 정보처리학과 전임강사,
2000.3 ~ 현재 울산대학교 디자인대학 정보디자인학과 조교수,
관심분야는 Knowledge Database, Data Mining, Multimedia Database.