울산대학교 조형논총 제4권 1호 pp. 99~112, 2001
Journal of Design Vol. 4, No. 1, pp. 99~112, 2001
University of Ulsan.

패스 표현의 효과적인 처리*

이 태 경
디자인학부 정보디자인전공

<요약>

최근 GIS, network, WWW, 멀티미디어 상연물등과 자료 변화의 순서가 중요한 영역의 문제들을 자연스럽게 표현할 수 있는 그래프 타입에 대한 연구가 이루어 지고 있다. 이 논문에서는 DAG(directed acyclic graph)로 표현되는 멀티미디어 상연물을 이용하여 DAG의 패스를 효과적으로 검색 처리하는 코드 시스템인 노드 코드(node code) 시스템을 소개한다. 노드 코스 시스템은 그래프의 각 노드마다 유일한 2진 문자열(binary string)을 부여한다. 두 노드 사이의 노드 코드의 비교하면 그래프 횡단 없이 두 노드가 연결되어 있느지를 확인할 수 있다. 노드 코드를 이용하여 두 노드 사이의 패스의 건설을 전통적인 그래프 횡단을 이용한 방법보다 효과적으로 수행하는 알고리즘도 소개한다.

키워드: 데이터베이스 시스템, 그래프 데이터 자료, 노드의 연결성, 노드 코드, 패스, 그래프횡단

Efficient Evaluation of Path Algebra Expressions

Lee, Tae-kyong
Dept. of Information Design, University of Ulsan

<Abstract>

Recently, there has been a lot of research on graph-type data because it can model the application domains such as GIS, network, WWW, multimedia presentations etc., and domains in which the data sequence is important. In this paper, an efficient code

* 이 논문은 2000년 울산대학교의 연구비에 의하여 연구되었음

system, called node code system, is proposed to evaluate paths of DAG in a multimedia presentation graph. The node code system assigns a unique binary string to each node of a graph. The comparison of node codes of two nodes tells the connectivity between the nodes. The method using the property of the node code system allows us to construct the paths between two nodes more efficiently than the method using conventional graph traversals. The algorithms to construct paths using the node code system are provided.

key words: DBMS, graph-type data, connectivity of nodes, node code, path, graph traversal

1. 서론

최근 비전통적인 타입의 자료들을 DBMS에 통합하는 문제에 많은 연구가 이루어지고 있으며 그 연구의 한 부분으로 그래프 타입에 대한 연구가 이루어지고 있다[1,7,8,12,14]. 그 이유로 그래프 타입은 GIS, network, WWW, 멀티미디어 상연물등과 같은 영역의 문제들을 자연스럽게 표현할 수 있으며 자료 변화의 순서(sequence)가 중요한 영역의 문제[9]들을 그래프의 패스를 이용하여 자연스럽게 표현하기 때문이다.

그래프 타입의 자료를 DBMS에 통합을 위해서는 다음과 같은 문제가 해결되어야 한다. 첫째, 각각의 그래프에 있는 노드가 가지고 있는 내용(contents)과 그래프의 에지로 표현되어 있는 각 노드 사이의 관계를 표현할 수 있는 질의 언어가 개발되어야 한다. 예를 들어 검색하고자 하는 노드가 가지고 있는 내용과 특정 내용을 가진 노드와 노드 사이가 관계가 인접되어 있느지, 연결되어 있는지 또는 같은 내용을 가진 노드가 연속으로 인접하고 있는지하는 노드들 사이의 관계를 표현할 수 있는 질의 언어가 필요하다. 둘째, 그 질의 언어를 처리할 수 있는 방법이 있어야 한다. 즉, 질의 언어로 표현된 특정 내용을 가진 노드들의 검색과 그 노드 사이의 관계를 만족하는 패스를 DBMS에서 검색, 처리하는 방법이 필요하다.

데이터베이스에서 그래프는 여러 상황(different contexts)에서 여러 연구자들의 관심이 되어 왔다. 그리고 이 그래프들을 표현하는 언어는 세가지로 분류할 수 있다. 첫째 시각적으로 노드 사이의 관계를 표현한 언어가 있다. 여기에 속하는 언어에는 G+[4,5], Hy+[2,3] GVISUAL[10]등이 있다. G+[4,5], Hy+[2,3]와 같은 시스템에서 데이터베이스는 그래프로 시각화(visualization)되고 질의는 데이터베이스를 대상으로 패턴을 비교하는 그래프의 집합으로 표시하며 노드들 사이의 관계를 나타내기 위하여 여러 종류의 에지를 사용한다. GVISUAL[10]은 노드 사이의 관계를 시간 연산자 next, until, eventually[6]에 해당하는 에지로 표현한다. 둘째, SQL과 비슷한(SQL-like) 모양을 가진 언어가 있다. 이에는 GraphDB[8], GOQL[10]이 있다. 그래프를 구성하는 노드, 에지, 패스등은 객체 지향의 클래스 개념을 사용하여 명시적으로 모델화 시킨다. 패스는 에지와 노드 그리고 패스 구성자(constructor)를 이용하여 표현한다. GraphDB[8]에서는 세가지의 클래스(simple classes, link classes, path classes)가 있다. 링크 클래스는 두 개의 심플 클래스를 "소스(source)"와 "타켓(target)"으로 정하여 연결한다. 패스 클래스는 링크 클래스를 이용한 정규식으로 표현된다. 즉, 패스

가 명시적으로 데이터 베이스 시스템안에 저장되어 있다. 검색을 위하여 "on...where...derive" 구절을 제공한다. 셋째, calculus에 바탕을 둔 언어이다. GCalculus/S[10,11]가 이 범주에 속하며 사용자 언어로 개발된 GVISUAL[10]의 formal basis로 개발된 언어이다. 당연히 패스 표현은 노드와 시간 연산자 next, until, eventually에 해당하는 연산자를 이용한다. GCalculus/S[10,11]는 O-Algebra[13]를 확대하여 대수적으로 처리된다.

그래프 타입의 자료 검색처리는 두 부분으로 나눌수 있다. 첫째, 사용자가 요구하는 특정한 내용을 가진 노드들을 검색하는 것이다. 둘째, 그 노드들 중에서 사용자가 요구하는 패스 관계를 만족하는 노드들로 구성된 패스를 검색하는 것이다. 이 논문에서는 첫째 부분은 이미 처리된 것으로 가정하고 특정한 내용을 가진 노드들이 질의에서 표현된 순서 관계를 만족시키는 패스를 검색처리하는 문제를 다룬다. 그리고 그래프 자료가 데이터 베이스에 저장되는 방법으로는 패스가 명시적으로 저장되어 있지 않고 부모 노드(parent node)와 자식 노드(child node)와의 관계만 저장되어 있고 질의에서 요구되는 패스는 그 정보로부터 만들어져 질의의 결과로 제공된다.

이 논문에서는 패스 표현을 위하여 GCalculus/S[10,11]를 이용하며 GCalculus/S[10,11]에서 사용된 연산자의 처리 방법을 다룬다. 따라서 이 논문에서의 패스에 관한 질의 처리 방법이 다른 질의 언어를 이용한 패스 표현을 처리할 수 있는가 하는 문제는 GCalculus/S[10,11]가 모든 종류의 패스를 표현할 수 있는가 하는 표현능력(expressive power)의 문제이다. [10]에 GCalculus/S와 정규표현(regular expression)과의 비교가 논의되어 정규표현과 GCalculus/S[10,11]와의 표현능력에는 거의 차이가 없음들 보여주고 있다. 이 결과는 다른 질의 언어가 GCalculus/S[10,11]로 변환(transformation)된다면 이 논문에서 사용되는 처리 방법은 다른 질의언어의 처리에도 사용될 수 있는 가능성이 있음을 보여준다. 이 변환 방법의 한가지 예로 [10]에 GVISUAL이 GCalculus/S로 변환되는 것을 보여준다.

2장에서는 GCalculus/S[10,11]를 간단하게 언급한다. GCalculus/S[10,11]를 설명하기 위하여 DAG(directed acyclic graph)의 모양으로 표현되는 멀티미디어 상연물[10,11]을 이용한다. 3장에서는 노드 사이의 연결성을 효과적으로 확인할 수 있는 코드 시스템인 노드 코드(node code)가 소개되며 노드 코드를 이용한 패스 처리 방법을 설명한다. 4장은 결론이다.

2. GCalculus/S

모든 언어는 데이터 모델에 바탕을 두고 있다. 이에 GCalculus/S[10,11]가 바탕을 두는 데이터 모델이 필요하다. 그리고 그래프 타입 자료의 응용 영역(application domain)으로 멀티미디어 상연물을 이용한다. 그 데이터 모델은 멀티미디어 자료들의 물리적 특징과 자료의 내용, 상연 그래프를 효과적으로 표현할 수 있어야 한다. 이에 객체 지향 데이터 모델을 이용하여 모델링을 한다. 이는 다음에 설명되는 이점이 있기 때문이다. 멀티미디어 자료에 나타나는 무수한 종류의 객체들과 그 객체들 사이의 관계는 class-subclass hierarchy와 composition hierarchy 이용하여 처리할 수 있다. 또한, 멀티미디어 자료의 물리적 특징인 분할화도 composition hierarchy를 이용하여 처리한다. 아래의 정의는 GCalculus/S[10,11]를 위한 데이터 모델이다.

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

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

class Pres_Edge
      [<Pres_Node>];
```

보기: {}, <>는 set constructor와 sequence constructor를 나타낸다.

```
class Pres_Node: inherits
      from Stream[
      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];
```

그림 2.1은 DAG의 형태로 표현되어 있는 멀티미디어 상연물이다. 노드는 스트림(stream)이라 불리며 멀티미디어 자료를 구성하는 최소 단위가 연속적으로 배치되어 구성된다. 예를 들어 비디오 스트림은 일련의 프레임(a sequence of frames)으로 구성된다. 각 자료들은 내용 객체(content object)와 객체들간의 관계(relationship)를 내포하며 스트림의 내용을 구성한다. 에지는 멀티미디어 상연물을 구성하는 노드들을 상연하는 방법을 표시한다.

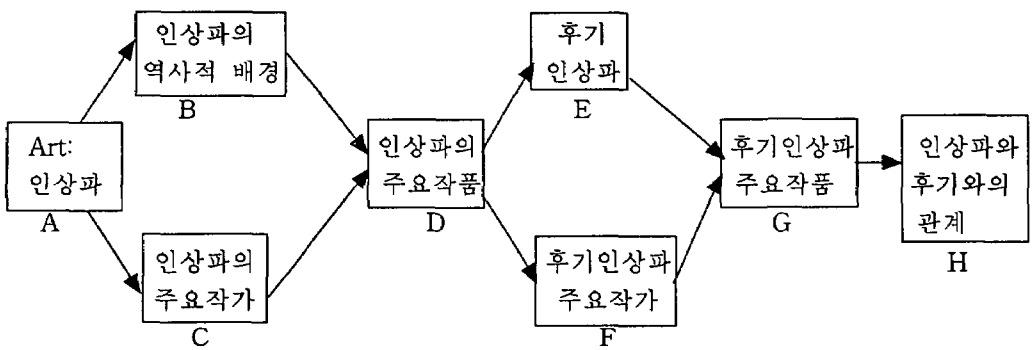

그림 2.1

그림 2.1의 멀티미디어 상연물은 인상파와 후기인상파를 설명한다. 인상파의 역사적 배경(노드 B)과 주요작가(노드 C)를 동시에 상연한후 주요작품(노드 D)를 상연한다. 노드 D의 상연후에 후기 인상파의 배경(노드 E)과 주요작가(노드 F)를 동시에 상연한다. 이후 주요작품(노드 G)과 인상파와 후기인상파의 관계(노드 H)를 상연한다.

GCalculus/S[]는 연산자 Next, Until, Connect를 이용하여 패스를 표현한다. 연산자 Next와 Until의 의미(semantics)는 temporal logic[8]에서 사용되는 연산자 next와 until의 의미와 유사하며 Connect는 eventually와 유사하다. 그림 2.1에 있는 패스가 어떻게 표현되는지 예를 이용하여 간단히 설명한다. GCalculus/S에 관한 상세한 내용은 []에 설명되어 있다.

Example 2.1 "후기 인상파의 주요작품" 노드와 인접한 노드로 이루어지는 모든 패스를 찾으시오.

{x| (∃g, ∃s1, ∃s2)(Pres_Graph(g) ⋀ Stream(s1) ⋀ Stream(s2) ⋀
<<g,x,s1>>[[s1]]s1.name = "후기 인상파의 주요 작품" **X**
[[s2]]*True*)}

패스는 일련의 노드이다. x는 검색하고자 하는 패스를 나타내는 변수(variable)이며 "**X**"는 Next에 해당하는 연산자이다. Next 연산자는 "바로 다음"의 의미를 가지고 있다. 위의 질의를 만족하는 패스는 두 개, <D,E>와 <D,F>이다.

Example 2.2 "인상파의 역사적 배경" 노드와 "후기 인상파 주요 작품" 노드 사이의 모든 패스를 찾으시오.

{x| (∃g, ∃s1, ∃s2)(Pres_Graph(g) ⋀ Stream(s1) ⋀ Stream(s2) ⋀
<<g,x,s1>>[[s1]]s1.name = "인상파의 역사적 배경" **C**
[[s2]]s2.name = "후기 인상파 주요 작품")}

"**C**"는 Connected에 해당하는 연산자이다. Connected 연산자는 두 노드 사이에 패스가 존재함을 주장(assertion)하는 연산자이다. 위의 질의를 만족하는 패스는 <B,D,E,G>와 <B,D,F,G>이다.

Example 2.3 인상파 화가 "Monet"가 계속 연속하여 나오다가 후기 인상파 "Cezanne"가 처음 나오는 패스를 찾으시오.

{x| (∃g, ∃s1, ∃s2)(Pres_Graph(g) ⋀ Stream(s1) ⋀ Stream(s2) ⋀
<<g,x,s1>>[[s1]](∃o1)(o1 ∈ s1.objects ⋀ o1.name = "Monet") **U**
[[s2]](∃o2)(o2 ∈ s2.objects ⋀ o2.name = "Cezanne"))

"**U**"는 Until에 해당하는 연산자이다. 스트림안에 객체(object)가 있다. 객체 "Monet"를 가지고 있는 연속적인 스트림(stream) 다음에 객체 "Cezanne"가 나오는 스트림으로 이루어지는 패스를 찾는 질의이다. 그림 2.1에서 노드 B, C, D가 객체 "Monet"를 가지고 있고 노드 E, F가 객체 "Cezanne"를 가지고 있다면 위의 질의를 만족하는 패스는 <B,D,E>, <B,D,F>, <C,D,E>, <C,D,F>이다.

3. 노드코드(Node Code)

하나의 소스 노드를 가지는 DAG에서 노드 사이의 패스의 유무를 판별할고 세 연산자를 처리할 수 있는 코드 시스템(code system)인 노드 코드(node code)를 소개한다. 만약 다수의 소스 노드를 가지고 있는 그래프의 경우에는 1개의 소스 노드를 첨가하여 원래의 소스 노드들을 첨가된 소스 노드의 자식 노드로 만들어 1개의 소스 노드를 가지는 그래프로 변형 시킨다. 이 코드는 2진수의 문자열(string)과 (노드 코드에 대한 설명의 편의를 위하여 2진수의 문자열을 사용한다. 그러나 당연히 이 2진수를 10진수로 바꾸어 사용할 수 있다.) 2진수의 문자열에서 실지로 유용한 정보를 가지고 있는 2진수의 길이에 관한 정보로 구성되어 있다. 모든 코드는 유일(unique)하며 각 노드는 소스 노드에서 각 노드까지의 패스의 수에 해당하는 코드의 수를 가진다. 두 개의 노드에 해당하는 노드 코드가 주어지면 서로 인접하고 있는지 (부모 자식간의 관계, 연산자 X), 또는 연결되어 있는지를 (패스가 존재하는지, 연산자 C) 노드 코드에 대한 단순 비교를 통하여 알 수 있으며 그 둘 사이의 패스는 실지의 그래프 횡단없이 만들 수 있다.

3.1 코드의 부여

노드 코드는 그래프를 depth-first-search로 횡단하여 각 노드에 일련의 2진 숫자와 2진 숫자에서 유용한 정보를 가지는 2진 숫자의 개수에 관한 정보이다. 그리고<a_0a_1 ... a_m, h> 모양으로 나타난다. ai의 값은 0 또는 1이며, $0 \le i \le m$, h의 값은 0과 m 사이의 숫자이다. 노드 코드의 a_0a_1 ... a_m 부분은 노드 번호, h 부분은 노드 번호의 유효 숫자(effective digit)라 부른다. m의 크기는 상응하는 2진 그래프에 있는 가장 긴 패스의 길이 보다 긴 어떤 수도 가능하다. 그리고 h의 값은 상응하는 2진 그래프의 소스 노드에서 각 노드까지의 패스의 길이와 일치한다. 소스 노드의 노드 코드로 <1_0 0_1 0_2 ... 0_m, 0>을 부여한다. 아래에 있는 Assign_Node_Code는 각 노드에 노드 코드를 부여하는 알고리즘이다. 처음 들어가는 input parameter는 (소스 노드, <1_0 0_1 ... 0_m, 0>)이다.

Algorithm: Assign_Node_Code_Binary(v, <a_1 a_2 ... a_m, h>)
Input: 노드 v, v의 노드 코드<a_0 a_1 ... a_m, h>
Output: v의 자식 노드들(child nodes)의 노드 코드
1 c = number of children of v (Ci는, $1 \le i \le c$, v의 자식 노드(child node))
2 for i = 1 to c do
3 begin
4 if (i = 1) then assign node code <a_0 a_1 ... a_h 0_{h+1} a_{h+2} .. a_m, h+i> to C_1
else assign node code <a_0 a_1 ... a_h 0_{h+1} 1_{h+2} .. 1_{h+i} $a_{h+(i+1)}$ .. a_m, h+i> to C_i;
5 Assign_Node_Code(Ci, <a_0 a_1 ... a_m, h>);
/* <a_0 a_1 ... a_m, h>는 Ci의 노드 코드 이다. */
6 end

알고리즘 Assign_Node_Code는 알고리즘 depth-first-search traversal 알고리즘을 이용한 것이다. 그림 3.1은 그림 2.1에 있는 DAG에 Assign_Node_Code_Binary를 적용하여 그

노드들에 부여 되는 노드 코드를 보여 주는 ordered tree이다. 어떤 노드는 1개 이상의 노드 코드를 부여 받는다. 각 노드의 노드 코드의 수는 소스 노드에서 그 노드 사이의 패스의 수를 나타낸다.

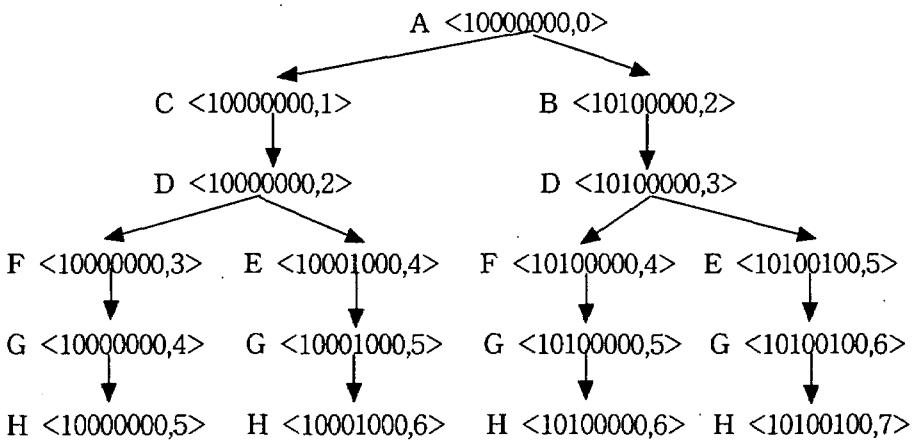

그림 3.1

지금까지 노드 번호의 설명을 위하여 m개의 bit을 이용한 2진수로 설명 하였다. 그러나 실지로 노드 번호를 10진법의 수로 바꾸어 사용할수 있다. 예를 들어 소스 노드의 노드 번호는 십진수 2m으로 변환된다. 알고리즘 Assign_Node_Code_Binary의 line 4의 if 부분에 해당하는 자식 노드(첫번째 자식)의 노드 번호는 변동이 없으나 else부분에 해당하는 두 번째 이후의 자식 노드들은 a_{h+2} .. a_{h+i}에 1이 부여 된다. 그러므로 부모 노드(parent node)의 노드 번호에 $2^{m-(h+2)}$, .. ,$2^{m-(h+i)}$를 더하면 자식 노드의 노드 번호에 상응하는 10진 수를 얻을수 있다. 이 경우를 위해서는 아래의 알고리즘 Assign_Node_Code_Decimal을 이용하고 소스 노드의 노드 코드는(소스노드, <2^m, 0>)이다.

```
Algorithm: Assign_Node_Code_Decimal(v, <N, h>)
Input: 노드 v, v의 노드 코드<N, h>
Output: v의 자식 노드들(child nodes)의 노드 코드
1 c = number of children of v(Ci는, 1 ≤i ≤c, v의 자식 노드(child node))
2 for i = 1 to c do
3 begin
4 if (i = 1) then assign node code <N, h+i> to C1
  else assign node code <N + 2^(m-(h+2)) + ... + 2^(m-(h+i)), h+i> to Ci;
5 Assign_Node_Code(Ci, <N, h>);
 /* <N, h>는 Ci의 노드 코드 이다. */
6 end
```

Remark 3.1: 노드 v의 노드 코드가 <N, h>라 하자. 그리고 v가 k개의 자식 노드를 가지면 자식 노드의 노드 코드는 첫번째 자식 노드부터 k번째의 자식노드까지 노드 번호는 차례로 $<N+0,\ h+1>$, $<N+0+2^{m-(h+2)},\ h+2>$, $<N+0+2^{m-(h+2)}+2^{m-(h+3)},\ h+3>$,...,$<N+0+2^{m-(h+2)}+2^{m-(h+3)}+\ldots+2^{m-(h+k)},\ h+k>$ 이 다 .

Proof: 노드 v의 노드 번호 N을 2진수로 표현하면 $a_1\ a_2\ \ldots\ a_h\ 0\ 0\ldots0_m$ 이다, $a_i = 0$ 또는 1, $1 \le i \le h$. 따라서 v의 첫번째 자식 노드부터 k번째의 자식노드까지 노드 번호는 $a_1a_2\ldots a_h0_{h+1}0\ldots0$, $a_1a_2\ldots a_h0_{h+1}1_{h+2}0\ldots0$, $a_1a_2\ldots a_h0_{h+1}1_{h+2}1_{h+3}0\ldots0$, ... , $a_1a_2\ldots a_h0_{h+1}1_{h+2}1_{h+3}\ldots1_{h+k}0\ldots0$ 이며 a_i의 값은 0또는 1이다. 이 노드 번호들을 10진수로바꾸면 $N+0, N+0+2^{m-(h+2)}$, $N+0+2^{m-(h+2)}+2^{m-(h+3)}$, ..., $N+0+2^{m-(h+2)}+2^{m-(h+3)}+\ldots+2^{m-(h+k)}$ 이다. 한편 자식 노드의 유효 숫자는 첫번째 자식 노드부터 k번째의 자식노드까지 h+1,...,h+k 이다.

Remark 3.2: 각각의 노드 코드는 유일하다. 즉, 노드 코드가 주어지면 상응하는 노드는 1개만 존재한다.

Proof: 한 노드의 노드 코드가 $<a_0a_1\ \ldots\ a_m,\ h>$라 하자. 그러면 $ai = 0$ or 1, $0 \le i \le h$, $aj=0$, $h+1 \le j \le m$, 이다. 이 노드의 처음 자식 노드의 노드 코드는 $<a_0\ \ldots\ a_h\ 0_{h+1}\ldots\ 0_m,\ h+1>$ 이며 i번째 자식 노드의 노드 코드는 $<a_0\ \ldots\ a_h\ 0_{h+1}\ 1_{h+2}\ \ldots\ 1_{h+i}\ 0_{h+i+1}\ \ldots\ 0_m,\ h+i>$ 이다. 즉, 한 노드의 모든 자식 노드의 노드 코드는 유일하다. 그러므로 모든 노드의 노드 코드는 유일하다.

Definition: 두 개의 노드 코드 $NC_1=<N_1, h_1>$, $NC_2=<N_2, h_2>$가 주어지면, $(N_1 > N_2)$ or $((N_1 = N_2)$ and $(h_1 > h_2))$이면 $NC_1 > NC_2$ 이다.

Remark 3.3: 노드 v의 노드 코드가 <N, h>라 하자. 그러면 v로부터 연결되어(connected) 있는 노드들의 노드 코드는 <N, h+1>과 같거나 크고 $<N+2^{m-(h+1)},\ h+1>$보다 작다.

Proof: 노드 v가 k개의 자식 노드를 가지고 있다고 가정하자. v의 첫번째 자식의 노드 코드는 <N, h+1>이다. 그리고 이 노드 코드는 v의 자손(descendant)들의 노드 코드 중에서 가장 적은 것이다. 왜냐하면 모든 자손들은 <N, h+1>보다 큰 노드 번호를 가지거나 큰 유효 숫자를 가지기 때문이다. 노드 v의 바로 오른쪽(첫번째) 형제 자매(sibling) 노드가 있으면 그 노드의 노드 코드는 $<N+2^{m-(h+1)},\ h+1>$이다. 그리고 v의 노드 코드를 $<a_0\ a_1\ \ldots\ a_h\ 0_{h+1}\ ..\ 0_m,\ _{h+i}>$으로 표현 한다면 v의 후손(descendant) 노드들의 노드 번호 a_{h+1}의 값은 0이다. 그러므로 v의 후손 노드들의 노드 번호의 값은 $N+2^{m-(h+1)}$보다 작다. 한편, v의 두번째 이후의 형제 자매 노드들의 노드코드는 $<N+2^{m-(h+1)},\ h+1>$보다 크며 그 형제 자매들의 후손 노드들의 노드 코드는 그 형제 자매 노드들의 노드 코드보다 크다. 그러므로 v로부터 연결되어(connected) 있는 노드들의 노드 코드는 <N, h+1>과 같거나 크고 $<N+2^{m-(h+1)},\ h+1>$보다 작다.

3.2 노드 코드를 이용한 연산자의 처리

노드 코드를 이용하여 세 연산자를 처리하는 방법을 설명한다. 가장 기본이 되는 처리 방법은 연산자 C를 처리하는 방법이다. 왜냐하면 연산자 X와 U는 C의 특수한 경우이기 때문이다. 즉, 인접성(adjacency, X)은 연결성(connectivity, C)을 만족하고 두 노드 사이의 관계가 부모 노드와 자식 노드이며 이루어지는 패스의 길이가 1인 경우이며 U는 두 노드 사이에 패스가 존재하고 그 패스의 마지막 노드를 제외한 노드가 같은 주장(assertion)을 만족 시키는 경우이다.

3.2.1 노드 코드 시스템을 이용한 C 연산자의 처리

이제 노드 코드 시스템을 이용하여 연산자 C를 처리하는 방법을 설명한다. 연산자의 처리란 검색 표현 (q C r)에 대하여 q와 r을 각각 만족시키는 subpath p1과 p2를 찾고 p1과 p2가 연산자 C의 의미(semantics)를 만족 하느가의 여부를 검사하여 p1과 p2로 구성되는 패스를 찾는 것이다.

q와 r을 만족 시키는 subpath는 한 개의 노드로 구성될수도 있으며 다수의 노드로 구성될수도 있다. 예를 들어 q가 (q_1 C q_2) C q_3)이고 q_i가 한 개의 노드에 관한 표현이면 q는 3개의 노드로 구성되어 있는 subpath에 대한 표현이다. 이에 q의 처리는 1): q_1과 q_2를 만족 시키는 노드들 중에서 C를 만족 시키는 path를 찾고 2): q_3를 만족시키는 노드를 찾고 3): 1)에서 찾은 패스의 마지막 노드와 3)에서 찾은 노드 사이에 C를 만족시키면 그 두 패스를 subpath로 하는 path를 찾는 것이다. 이에 앞으로 일반성을 상실함이 없이 설명의 편의를 위하여 q와 r을 만족 시키는 subpath는 한 개의 노드로 구성된 패스로 한정한다. 왜냐하면 q의 처리는 qi를 검색하는 방법을 제외하면 연산자 C처리방법의 연속적인 적용이기 때문이다. 또한 이 논문의 주제가 아닌 q_i를 만족하는 노드의 검색 방법은 주어진 것으로 가정하고 q와 r을 만족하는 subpath는 주어진 것으로 가정한다.

검색 표현 (q C r)에서 q와 r을 만족 시키는 노드가 각각 v_1과 v_2라 하자. 노드 v_1이 v_2에 연결성(connectiveness)의 확인은 Remark 3.3을 이용한다. v_1과 v_2의 노드 코드가 각각 $<N_1, h_1>$, $<N_2, h_2>$라 하자. Remark 3.3에 의해 v_1에 연결되어 있는 노드의 노드코드는 $<N_1, h_{1+1}>$과 같거나 크고 $<N_1+2^{m-(h+1)}, h_1+1>$보다 작아야 한다. 만약 v_2의 노드코드가 위의 조건을 만족 시키지 못하면 v_2가 v_1에 연결되어 있을 가능성은 없다. 만약 위의 조건을 만족 시키면 v_1의 자식 노드들의 노드 코드들과 v_2의 노드 코드를 이용하여 연결성을 검사한다. 자식 노드들중에서 가능성이 없는 노드들은 제외 시키고 가능성이 있는 노드들만 뽑는다. 그 노드들과 v_2와의 연결성 검사를 노드 코드를 가지고 계속한다. 이러한 과정을 v_2에 도달할 때 까지 계속한다.

Algorithm: Make_Connect(v_1, v_2)
Input: 두 개의 노드 v_1, v_2
Output: 처음 노드는 v_1, 마지막 노드는 v_2인 패스
1 h_1 = v_1의 노드 길이; h_2 = v_2의 노드 길이;

2 N_1 = v_1의 노드 번호; N_2 = v_2의 노드 번호;
3 Initialize S to be empty; /* S is a stack. */
4 Initialize l to be empty; /* l is a list storing nodes making up a path connecting v_1 to v_2. */
5 Initialize L to be empty; /* L is a list of lists to store possibly more than one paths connecting v_1 to v_2.. */
5 if ($<N_1, h_1+1> \leq <N_2, h_2> < <N_1+2^{m-(h1+1)}, h_1+1>$), then begin
6 push(S, v_1);
7 **while** ($S \neq$ empty) **do begin**
8 v = Pop(S);
9 **if** (v = v_2) **then begin**
10 AddEnd(l, v); /* path의 마지막 노드 v_2를 l에 더함. */
11 AddEnd(L, l); /* 확인된 path l을 L에 더함. */
12 **if** ($S \neq$ empty) **then begin**
13 v = Pop(S);
14 **while** ((v $\neq$ a child node of l.Last) $\bigwedge$ (l $\neq$ empty)) **do**
15 RemoveEnd(l); /* a path를 확인한 뒤 가능한 다른 path를 확인하기 위하여 가장 나중에 branch한 노드중 모든 child node가 search 되지 않은 노드까지 backtrack. */
16 Push(S, v);
endif
endif
17 **else begin**
18 N = v의 노드 번호; h = v의 노드 길이;
19 **if** ($<N, h+1> \leq <N2, h2> < <N+2^{m-(h+1)}, h+1>$), **then begin**
20 AddEnd(l, v);
21 **for** every child node v_c of v **do** Push(S, v_c)
endif
endelse
21 **endwhile**
22 **endif**

알고리즘 Make_Commect을 이용하면 v_1을 포함해서 노드 v_1과 v_2로 만들어 지는 패스 상에 있는 노드들을 제외한 노드들은 Remark 3.3에 의하여 search 되는 불필요한 과정이 없어진다는 것이다. 그림 3.2를 가지고 설명하면 처음 v_1의 자식 노드 v_4와 v_5에 연결되어 있는 노드들로 구성되어 있는 서브그래프(subgraph)들을 횡단하는 과정을 피하며 v_3에 연결된 노드들 중에서도 v_1과 v_2 사이의 노드들을 제외한 노드들을 횡단하는 과정을 피하게 된다. 예를 들어 건설하고자 하는 패스 상에 있는 노드 vi에 연결되어 있는 노드 v_5와 v_6에 대한 횡단도 제외된다.

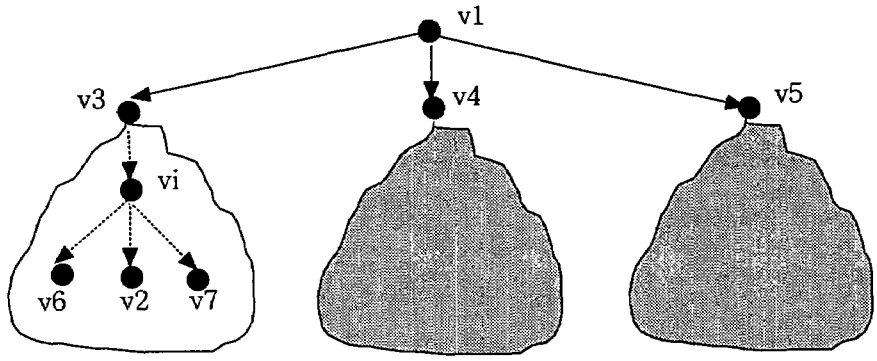

그림 3.2

위의 설명된 알고리즘의 performance 분석한다. 알고리즘의 complexity는 두 노드가 주어질 때 두 노드 사이의 패스를 구축하기 위하여 방문하는 노드의 수로 나타낸다. n개의 노드가 있으며 각각의 노드는 c1개의 노드 코드를 가지고 c_2개의 자식 노드를 가진다고 가정하자. h를 그래프의 높이(height)라고 하면 $1+c_2+ \ldots +c_2^{h-1}=n$. $h=\log c_2^{n(c2-1)+1}$. 따라서 알고리즘 Make_Connect를 이용하면 방문하는 노드의 수는 최대 $(c1 \times c2 \times \log c_2^{n(c2-1)+1})$이다. 이것은 $O(\log c_2^n)$이다. 이 결과는 전통적인 방법을, depth-first search나 breadth-first search, 사용할 때 방문하는 노드의 최대수 $O(n)$보다 좋은 결과이다.

노드 번호의 유효 숫자를 이용하면 더욱 효율적인 패스 건설을 할 수 있다. 노드 vi와 vj의 노드 코드가 각각 $<a_0a_1 \ldots a_i \ldots a_m, i>$, $<a_0a_1 \ldots a_i\ a_{i+1} \ldots a_j \ldots a_m, j>$, $j>i$, 이며 vj가 vi의 자식 노드라 하자. vj는 vi의 첫 번째 자식 노드이거나 두 번째 이후의 자식 노드가 될 수 있다. 만약 vj가 vi의 첫 번째 자식 노드라면 a_j의 값은 0이며 a_k의, $j+1 \le k \le$, 0이다. 어떤 노드의 첫 번째 자식노드의 노드 번호는 유효숫자의 마지막 순서에 해당하는 2진수와 그 이후에 나오는 2진수의 값이 0이며 유효숫자 마지막 순서의 이전 값들은 부모 노드의 노드 번호화 일치한다. 즉, 이 경우 부모 노드의 노드 코드는 $<a_0a_1 \ldots a_i\ 0_j \ldots 0_m, j-1>$ 이다. 만약 vj가 두 번째 이후의 자식 노드인 경우 vj의 노드 코드는 $<a_0a_1 \ldots a_i\ 0_{i+1}\ 1_{i+2} \ldots 1_j\ 0_{j+1} \ldots 0_m, j>$ 이다. 이는 vj가 vi의 (j - i)번째 자식 노드임을 보여준다. 이는 vj가 유효 숫자의 마지막 자리에서 시작하여 왼쪽으로 이동하여 처음 0이 나오는 순서의 이동 수의 크기만큼 부모 노드에서 떨어져 있음을 보여준다. 이 경우 vj가 vi의 k번째 자식 노드이면 부모 노드의 노드 코드는 $<a_0a_1 \ldots a_i\ 0_{i+1}\ 0_{i+2} \ldots 0_j\ 0_{j+1} \ldots 0_m, j-k>$ 이다. 이 설명은 Remark 3.1을 달리 설명한 것이다.

어떤 특정한 노드의 노드 코드를 이용하여 부모 노드의 노드 코드를 알아내는 방법을 두 노드 사이의 패스 건설에 사용할 수 있다. 알고리즘 Make_Commect는 노드 v1에서 v2까지의 패스건설을 v1에서 시작하였다. 이와는 반대로 패스의 마지막 노드 v2에서 시작하면 패스 상에 있는 노드의 부모 노드를 노드 코드를 이용하여 찾을수 있기 때문에 패스 상에 있지 않는 노드를 방문할 필요가 사라진다. 이는 패스상에 있을 가능성이 있는 노드들을 stack에 push하고 다시 Remark 3.3의 성질을 이용하여 실지로 패스상에 존재하는지 하는 검사하는 부분을 제거 시킨다. 물론 big-oh 표현으로는 complexity상으로 차이가 없다.

3.3 노드 코드 시스템을 이용한 X와 U 연산자의 처리

검색 표현 (q X r)에서 q와 r을 만족 시키는 노드가 각각 v_1과 v_2라 하자. 노드 코드를 이용하여 연산자 X의 처리는 Remark 3.1을 이용한다. v2가 v1의 자식 노드라면 v2 노드 코드의 유호 숫자가 v_1 노드 코드의 유효 숫자보다 커야 한다. 이 조건을 만족 시키는 경우 v_1의 노드 코드의 유효 숫자가 h1, v2의 노드 코드 유효 숫자가 h2라 하자. 이는 v2가 v1의 (h2 - h1)번째 자식 노드이어야 한다. 이제 v2가 v1의 (h2 - h1)번째 자식 노드라는 정보를 가지고 v1의 노드 코드를 계산할 수 있으며 이것이 실지의 v1 노드코드와 비교하여 v1과 v2가 실지로 부모, 자식 노드의 관계이면 패스를 건설한다. 이를 위한 알고리즘의 complexity는 $O(c)$이다.

검색 표현 (q U r)에서 q를 만족 시키는 노드는 { v_{11}, v_{12}, ... , v_{1n} }이며, $n \geq 1$, r을 만족 시키는 노드는 v_2라 하자.(물론 r을 만족 시키는 노드가 1개 이상 가능하다. 이 경우에는 위의 검색 표현을 만족 시키는 패스가 1개 이상일 가능성이 있는 경우이다. 여기에서는 설명의 편의를 위하여 1개의 노드만 가정 하였다.) v2에서 v2의 노드 코드를 이용하여 v2의 부모 노드의 노드 코드를 찾고 이 노드 코드를 가지는 노드가 { v_{11}, v_{12}, ... , v_{1n} }에 존재 하는지 확인한다. 다음에 이 부모 노드의 부모 노드가 { v_{11}, v_{12}, ... , v_{1n} }에 존재하는지 확인한다. 이 과정을 마지막으로 찾은 부모 노드의 부모 노드가 { v_{11}, v_{12}, ... , v_{1n} }에 존재하지 않거나 { v_{11}, v_{12}, ... , v_{1n} }에 있는 모든 노드가 방문되었다면 방문된 순서의 역순으로 나열된 노드가 (q U r)를 만족하는 패스이다. 이를 위한 알고리즘의 complexity는 $O(\log_{c2}{}^{n})$이다.

4. 결론

이 논문에서 노드 코드 시스템을 이용하여 DAG의 패스에 대한 질의에 관한 검색 처리를 효과적으로 처리하는 방법을 제시하였다. 그리고 그 응용 영역으로 멀티미디어 상연물을 가지고 설명하였다. 노드 코드는 그래프의 소스 노드에서 각 노드에 이르는 패스를 유일하게 확인할 수 있는 코드를 각 노드에 부여한다. 만약 소스 노드에서 어떤 노드에 이르는 패스가 1개 이상이면 그 노드는 패스에 상응하는 수의 코드를 부여받는다. 두 노드의 연결성은 그래프 횡단없이 두 노드의 비교로 확인할 수 있다. 그리고 두 노드 사이의 패스의 건설도 그래프 횡단없이 노드 코드를 이용하여 건설할 수 있다. 그래프 횡단이 없는 패스의 건설은 전통적인 횡단을 이용한 "path-at-a-time"의 방법보다 좀 더 효율적인 "set-at-a-time"을 이용한 패스의 건설을 가능하게 한다.

노드 코드의 성질을 이용한 두 노드 사이의 패스 건설을 위한 알고리즘을 제시하였다. 이 알고리즘은 최악의 경우의 complexity가 $O(\log_c{}^{n})$ 이며 c는 각 노드가 가지는 자식 노드의 수에 대한 가정이며 n은 노드의 수이다. 이 complexity는 전통적인 횡단 알고리즘의 complexity인 $O(n)$보다 효율적이다.

이 논문은 응용영역으로 DAG로 모델화 할 수 있는 자료를 이용하였다. 노드 코드를

Undirected Graph에 적용하더라도 노드 코드의 성질을 이용할 수 있다. 그러나 일반적인 그래프의 형태인 Undirected Cyclic Graph에는 노드 코드의 특징을 이용할 수가 없다. 이 문제에 대한 연구가 앞으로 필요하다. 응용 영역으로 사용한 멀티미디어 상연물 그래프의 경우는 sparse graph이다. 노드의 개수가 많은 그래프의 경우 노드 번호의 크기가 커진다. 이에 노드 번호의 크기를 조절할 수 있는 방법에 대한 연구가 필요하다.

참고문헌

1. Biskup, J., U. Rasch, and H. Stiefeling, "An Extension of SQL for QueryingRelations", Computer Languages 15(1990), pp 65 - 82.
2. Consens, M., Mendelzon, A., "GraphLog: A Visual Formalism for Real-Life Recursion", ACM PODS Conference, 1990.
3. Consens, M., Mendelzon, A., "Hy+: A Hygraph-based Query and Visualization System", ACM SIGMOD Conference, 1993.
4. Cruz, I.F., Mendelzon, A., Wood, P.T., "A Graphical Query Language Supporting Recursion", ACM Sigmod Conference, 1987
5. Cruz, I.F., Mendelzon, A., Wood, P.T., "G+: Recursive Queries witnhour recursion", 2nd Int. Conference on Expert Database Systems, 1988
6. Emerson, E., "Temporal and Modal Logic" in Handbook of Theoretical Computer Science, Chapter 16, Leeuwen J., editor, pp 995 - 1072, Elsevier, 1990
7. Gyssens, M., Paredaens, J., Bussche, J., Gucht D., "A Graph-Oriented Database Model", IEEE Trans. on Knowledge and Data Engineering, vol. 6, No. 4, Aug. 1994, pp 572 - 586
8. Guting, R., "GraphDB: Modeling and Querying Graphs in Databases", VLDB, conf., 1994, pp 297 - 308
9. Lee, Taekyong, Bozkaya, T, Kuo, H-C, Özsoyoğlu, G. and Özsoyoğlu, Z. M., "A Scientific Multimedia Database System for Polymer Science Experiments", Eighth International Conference on Scientific and Statistical Database Systems, 1996, pp 86 - 95
10. Lee, Taekyong, Sheng, L., Bozkaya, T., Balkir, N.H., Ozsoyoglu, G., Ozsoyoglu, Z.M., "Querying Multimedia Presentations Based on Content", IEEE Trans on Knowledge and Data Engineering, vol. 11, No. 3, Jun., 1999, pp 361 - 385
11. 이태경, "멀티미디어 상연그래프 질의언어와 대수를 이용한 질의처리방법", 정보과학회 논문지, 2000년 6월, 27권 2호, pp 185 - 198
12. Levene, M., Loizou, G., "A Graph-Based Data Model and its Ramifications", IEEE Trans. on Knowledge and Data Engineering, vol. 7, No. 5, Oct., 1995, pp 809 - 823
13. Lin, J., Ozsoyoglu, Z.M., "Processing OODB queries by O-Algebra" CIKM, Nov. 12 - 16, 1996, Rockville, MD. pp 134 - 142
14. Poulovassilins, A., Levene, M., "A Nested-Graph Model for the Representation and Manipulation of Complex Objects", ACM Trans on Information Systems, Vol. 12, No. 1, Jan. 1994, pp 35-68