# 가지형 패턴의 시퀀스화를 이용한 XML 문서 필터링

## (FiST: XML Document Filtering by Sequencing Twig Patterns)

권 준 호 † Praveen Rao †† 문 봉 기 †† 이 석 호 †††
(Joonho Kwon) (Praveen Rao) (Bongki Moon) (Sukho Lee)

**요 약** 최근 XML 문서 필터링에 기반한 출판-구독(publish-subscribe) 시스템이 많은 관심을 받고 있다. 전형적인 출판-구독 시스템에서, 구독자들은 XPath 언어로 명세된 프로파일로 자신들의 관심을 표현하고, 새로운 내용들은 사용자 프로파일에 대하여 매칭 여부를 판단하여 관심을 가지고 있는 사용자들에게만 배달된다. 구독자의 수와 그들의 프로파일이 증가할수록, 시스템의 확장성이 출판-구독 시스템의 중요한 성공 요소가 된다. 이 논문에서는 XPath로 명세된 가지형 패턴과 입력 XML 문서들을 Prüfer의 방법을 사용하여 시퀀스로 변환하는 FiST라 불리는 새로운 필터링 시스템을 제안한다. FiST 시스템은 가지형 패턴을 구성하는 선형 경로들에 대하여 각각 매칭을 수행하고 후처리 과정에서 그 결과들을 병합하는 방법을 이용하는 대신에 가지형 패턴 전체를 사용하여 입력 문서에 대하여 매칭을 수행한다. 또한 효율적인 필터링을 위하여 시퀀스들을 해시 기반의 동적 인덱스로 구성한다. 실험 결과를 통해 전체 매칭 접근 방법이 다양한 환경에서 낮은 필터링 비용과 좋은 확장성을 가짐을 알 수 있다.

**키워드** : XML 문서 필터링, 가지형 패턴, Prüfer 시퀀스

***Abstract*** In recent years, publish-subscribe (pub-sub) systems based on XML document filtering have received much attention. In a typical pub-sub system, subscribing users specify their interest in profiles expressed in the XPath language, and each new content is matched against the user profiles so that the content is delivered only to the interested subscribers. As the number of subscribed users and their profiles can grow very large, the scalability of the system is critical to the success of pub-sub services. In this paper, we propose a novel scalable filtering system called **FiST**(**F**iltering by **S**equencing **T**wigs) that transforms twig patterns expressed in XPath and XML documents into sequences using Prüfer's method. As a consequence, instead of matching linear paths of twig patterns individually and merging the matches during post-processing, FiST performs **holistic matching** of twig patterns with incoming documents. FiST organizes the sequences into a dynamic hash based index for efficient filtering. We demonstrate that our **holistic matching** approach yields lower filtering cost and good scalability under various situations.

**Key words** : XML document filtering, twig pattern, Prüfer sequence

## 1. 서 론

출판-등록(publish-subscribe) 시스템은 선택적 정보 전송(selective dissemination of information) 덕분에 e-commerce와 인터넷 애플리케이션에서 중요한 역할을 하고 있다. 전형적인 출판-등록 시스템은 새로운 내용이 생성될 때마다 그 내용을 관심을 가지고 있는 구독자들에게만 전송한다. 이러한 서비스들은 수백만의 등록된 사용자들에 대하여 많은 수의 내용들을 효율적으로 매칭하는 소프트웨어 시스템을 필요로 한다.

정보 교환을 위한 표준으로 자리 잡은 XML(extensible markup language)의 인기는 정보 전달(information dissemination)을 위한 XML 필터링 시스템을 개발하려는 다양한 연구를 촉진시켰다. 일반적으로 XML 필터링 시스템에서는 사용자들이 XPath 언어[1]를 사용하여 사용자 프로파일을 명세한다. 이 논문에서는 **가지**

† 학생회원 : 서울대학교 전기컴퓨터공학부
bluerain@db.snu.ac.kr
†† 비 회 원 : University of Arizona Department of Computer Science
rpraveen@cs.arizona.edu
bkmoon@cs.arizona.edu
††† 종신회원 : 서울대학교 전기컴퓨터공학부
shlee@snu.ac.kr
논문접수 : 2006년 1월 25일
심사완료 : 2006년 5월 2일

**형 패턴(twig pattern)**으로 표현되는 사용자 프로파일을 다룬다. 가지형 패턴은 자식(child)과 후손(descendant) XPath 축(axis)을 포함하는 2개 이상의 선형 경로(linear path)로 구성된다. 다음과 같은 XPath 식이 가지형 패턴이다.

book[author//name="John"]/title

XML 필터링 문제는 XML 문서에서 모든 가지형 패턴의 출현을 발견하는 문제와는 다른 문제이다. XML 문서와 가지형 패턴의 역할이 정반대로 동작하기 때문이다. 이 논문에서 풀고자 하는 필터링 문제는 다음과 같이 표현할 수 있다.

가지형 패턴들의 집합 Q와 XML 문서 D가 주어질 때, 모든 $q \in Q'$가 문서 D와 매칭하는 서브셋 $Q' \subseteq Q$ 을 찾아라.

이 논문에서는, 입력 XML 문서에 대하여 가지형 패턴들의 전체 매칭(holistic matching) 여부를 수행하는 FiST(**Fi**ltering by Sequencing **T**wigs) 시스템을 제안한다. FiST 시스템은 가지형 패턴을 여러 개의 선형 경로들로 분해하여 처리하지 않기 때문에 전체 매칭이라고 할 수 있다. FiST는 순서 가지형 패턴 매칭(ordered twig pattern matching) 문제에만 집중하는데, 순서 매칭 문제는 Muller의 최근 연구[2]와 같이 가지형 패턴의 노드가 XML 문서에서 나타나는 순서를 따라야 하는 응용에서 매우 중요하다.

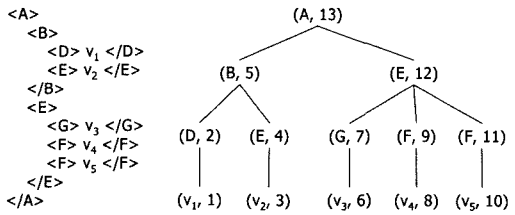
FiST 시스템은 XML 문서와 가지형 패턴들을 Prüfer 시퀀스[3]로 변환하여 전체 가지 패턴 매칭(holistic twig pattern)을 수행한다. 가지형 패턴을 변환하여 얻은 시퀀스들은 효율적인 필터링을 위해 해시 기반의 동적인 인덱스에 저장된다. FiST의 매칭 알고리즘은 점진적인 서브시퀀스 매칭 단계와 브랜치 노드 검증 단계의 두 단계로 구성된다. 첫 번째 단계에서 입력 XML 문서와 매칭하는 가지형 패턴들의 수퍼셋(superset)을 판별하고, 두 번째 단계에서 수퍼셋에 속한 가지형 패턴들 중에서 브랜치 노드에 대한 후처리 과정을 통해 잘못된 결과들을 추려낸다. 실험을 통하여 전체 가지형 패턴 매칭 접근 방법을 이용한 FiST 시스템이 다양한 상황 아래에서 좋은 확장성을 가지고 있음을 보였다.

이 논문의 구성은 다음과 같다. 2장에서 제안하는 기법의 배경과 동기를 설명한다. 3장에서는 FIST 시스템의 구조를 설명한다. 4장에서는 FiST 시스템의 핵심 알고리즘에 대하여 설명한다. 5장에서는 실험 결과에 대하여 분석하고, 마지막으로 6장에서는 연구 결과를 정리하고 결론을 맺는다.

## 2. 배경과 연구 동기

### 2.1 XML 문서와 Prüfer 시퀀스

XML 문서는 순서를 가진 레이블된 트리로 모델링할 수 있다. 그림 1(a)의 XML 문서는 그림 1(b)처럼 순서를 가진 레이블된 트리로 표현할 수 있다. 트리에 있는 각각의 노드는 XML 문서의 엘리먼트나 값(value)에 해당한다. 값들은 문자 데이타(CDATA, PCDATA)로 표현되고 리프 노드에 나타난다. 트리의 간선은 2개의 엘리먼트 사이의 관계나 엘리먼트와 값 사이의 관계를 표현한다. 각각의 엘리먼트는 (애트리뷰트, 값) 쌍을 가질 수 있다. 애트리뷰트는 엘리먼트와 같은 방식으로 처리할 수 있으므로, 이 논문에서는 애트리뷰트와 엘리먼트를 따로 구분하지 않는다.




(a) XML 문서 (b) 트리 표현

그림 1 샘플 XML 문서

Prüfer(1918)는 트리에서 한 번에 하나의 노드를 삭제하여 레이블된 트리와 시퀀스 사이의 일대일 대응을 이루는 방법을 제안하였다[3]. 1부터 n을 레이블로 가진 n개의 노드를 가지고 있는 트리 $T_n$에서 시퀀스를 생성하는 알고리즘은 다음과 같다. $T_n$에서 가장 작은 번호를 가진 리프 노드를 삭제하고, 이를 $T_{n-1}$이라 한다. 삭제된 노드의 부모 노드가 가진 레이블을 $a_1$이라고 하자. $T_{n-1}$에서 가장 작은 번호를 가진 리프 노드를 삭제하여 $T_{n-2}$를 얻고, 그 노드의 부모 노드가 가진 레이블 $a_2$라 하자. 하나의 간선으로 연결된 2개의 노드가 남을 때 까지 이 동작을 반복한다. 추출된 시퀀스 $<a_1, a_2, a_3, ..., a_{n-2}>$가 트리 $T_n$의 Prüfer 시퀀스이다. 추출된 시퀀스로부터 원래의 트리 $T_n$을 재구성할 수 있다. 트리 $T_n$의 Prüfer 시퀀스의 길이는 n-2이다.

PRIX 시스템[4]과 같은 방식으로 오직 하나의 노드가 남을 때까지 트리 노드의 삭제를 계속하면 길이가 n-1인 Prüfer 시퀀스를 얻을 수 있다. 또한 XML 문서의 레이블된 Prüfer 시퀀스(LPS)는 노드 번호 대신 태그 이름(엘리먼트)을 치환하면 얻을 수 있다[4]. 문서 트리의 리프 노드에 더미 자식 노드를 추가하면 **확장 Prüfer 시퀀스**를 생성할 수 있다. 확장 Prüfer 시퀀스에는 원래 트리의 리프 노드 레이블들도 나타난다. 아래 예제는 XML 문서 트리를 위한 시퀀스를 설명한다.

**예제 1.** 그림 1(b)에 있는 XML 문서를 고려하자. 트리의 노드는 후위순회 순서로 레이블 되어 있다. 노드

번호를 이용한 XML 문서의 Prüfer 시퀀스는 2 5 4 5 13 7 12 9 12 11 12 13이 된다. 태그 이름을 이용한 LPS는 D B E B A G E F E F E A가 된다. 문서의 리프 노드 $v_1$부터 $v_5$에 가상의 자식 노드가 있다고 생각하고 태그 이름을 이용하면, 확장 LPS는 $v_1$ D B $v_2$ E B A $v_3$ G E $v_4$ F E $v_5$ F E A가 된다.

이 절 이후, 이 논문에서 나오는 Prüfer 시퀀스는 확장 Prüfer 시퀀스를 의미한다.

### 2.2 관련 연구

기존의 XML 필터링 연구로서 다양한 연구가 진행되었다. XFilter[5]는 XML 필터링 문제에서 첫 번째 연구로서 선형 경로(linear path)만 존재하는 XPath 식을 유한 상태 기계(finite state machine)로 변환하여 입력 XML 문서와의 매칭 여부를 판단하였다. 후속 연구인 YFilter[6]는 더 좋은 확장성을 위해서 XPath 식의 공유 처리 방법을 제안하였고, 사용자가 명세한 모든 XPath 식을 하나의 비결정적 오토마타(non-deterministic finite automata)로 구성하여 이용한다. YFilter는 가지형 패턴을 여러 개의 선형 경로로 분해하고 그 각각의 선형 경로에 대하여 매칭 여부를 확인한다. 매칭된 선형 경로들이 가지형 패턴 조건을 만족하는지 알기 위해서 후처리 작업을 수행한다. book[author//name]/title과 같은 XPath 식을 예로 들면, YFilter는 이 XPath를 2개의 선형 경로 book/title과 book/author//name으로 분해하고 하나의 NFA로 구성한다. 가지형 패턴 매칭을 위해서 각각의 선형 경로에 대한 매칭을 여부를 확인한 후, 선형 매칭들이 가지형 패턴을 이루는지 확인하는 후처리 과정을 수행한다. 이에 반하여 FiST 시스템은 가지형 패턴과 입력 문서를 Prüfer 시퀀스로 변환하여 전체 가지형 패턴 매칭(holistic twig pattern matching)을 수행한다. FiST와 YFilter는 하나의 실행시간 스택을 사용하지만, FiST는 현재 처리되고 있는 태그부터 루트에 이르는 경로에 있는 태그들을 스택에 저장한다. 스택의 크기는 문서의 최대 크기에 좌우된다. 반면에 YFilter는 NFA의 수행 동안에 현재 태그가 처리되기 이전의 활성화 상태를 추적하기 위하여 스택을 사용한다.

복잡한 가지형 패턴의 필터링을 제공하기 위하여 XTrie[7]라 불리는 트라이 기반 자료 구조가 제안되었다. XTrie도 가지형 패턴을 선형 경로들로 분해하고 각 선형 경로에 대하여 매칭을 수행한다. 오토마타에 버퍼를 사용한 필터링 연구들도 수행되었다. XPush[8]는 프리디킷을 처리하기 위해서 변형된 결정적 푸시다운 오토마타(deterministic pushdown automata)를 사용하였다. Bruno 등은 인덱스-기반과 탐색 기반 XML multi-query 처리[9]를 연구하였으며, 그 각각이 장단점이 있음을 보였다. Lazy DFA[10]는 많은 수의 XPath 질의로부터 레이지(lazy)하게 생성한 하나의 결정적 오토마타(deterministic finite automata)가 적은 수의 상태를 가지면서 효과적으로 XML 필터링에 사용될 수 있음을 보였다. 최근에는 관계 데이타베이스 시스템에 기반한 XML 기반의 출판-구독 시스템을 제안한 연구도 있었다[11].

### 2.3 연구 동기

이전 연구 중에서 입력 XML 문서에 대하여 XPath 문법을 따르는 가지형 패턴의 전체 매칭을 지원하는 필터링 시스템이 없었다. 또한 XML에서 문서에 나타나는 엘리먼트들의 순서를 지켜야하는 응용에서 필요한 순서 가지형 패턴 매칭 문제에 대한 연구도 없었다. 이런 점들을 해결하기 위해서 순서 가지형 패턴 매칭을 지원하면서 전체 가지형 패턴 매칭을 지원하는 필터링 시스템을 제안하게 되었다.

## 3. FiST 시스템

### 3.1. XML 문서 필터링

XML 문서 필터링 문제는 XML 데이타베이스에서 모든 가지형 패턴을 찾는 것과는 다른 동작을 필요로 한다. 전통적인 XML 인덱싱과 질의 처리 문제에서는 문서에 나타나는 모든 가지형 패턴을 빨리 찾기 위하여 XML 문서를 인덱싱한다(XISS[12], TwigStack[13], PRIX[4]). 그러나 XML 필터링에서는 가지형 패턴과 XML 문서의 역할이 그 반대이다. XML 필터링 문제에서는 입력 문서에 어떤 가지형 패턴이 나타나는지 여부를 빨리 결정하기 위해서 가지형 패턴들을 인덱싱한다. XML 문서 필터링 문제를 다음과 같이 정의할 수 있다.

> 가지형 패턴들의 집합 Q와 XML 문서 D가 주어질 때, 모든 $q \in Q'$가 문서 D와 매칭하는 서브셋 $Q' \subseteq Q$ 을 찾아라.

이 논문에서는 가지형 패턴에서의 노드들의 순서가 문서에서의 노드들의 순서와 일치하는 순서 매칭만을 다룬다.

### 3.2 FiST 시스템의 구조

그림 2는 FiST 시스템의 핵심 구성 요소와 전체적인 구조를 보여준다. XPath로 명세한 사용자 프로파일을 XPath 파서가 파싱하여 Prüfer 시퀀스로 변경한다. 이 부분은 3.3.절에서 설명한다. 필터링 엔진의 동작 중에 사용자 프로파일은 갱신될 수 있다. 변경된 시퀀스들은 시퀀스 인덱스(Sequence Index)라 불리는 해시 기반의 동적 인덱스에 저장된다. 각각의 시퀀스들은 프로파일 시퀀스(Profile Sequence)라 불리는 리스트 형태로 유지된다. 이 시퀀스 인덱스와 프로파일 시퀀스가 필터링 엔진의 핵심 자료 구조이며, 자세한 내용은 4장에서 설명한다.

SAX 파서[14]가 입력으로 주어지는 XML 문서를 파싱한다. SAX 파서는 엘리먼트의 시작 태그마다 start tag 이벤트를 생성하고, 엘리먼트의 끝 태그마다 end tag 이벤트를 생성한다. 필터링 엔진은 점진적으로 문서로부터 Prüfer 시퀀스를 구성하면서 SAX 파서의 이벤트마다 특정한 동작을 수행한다.

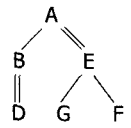
## 4. 인덱스 구조와 필터링 알고리즘

이 장에서는 FiST의 인덱스 구조와 필터링 알고리즘에 대하여 설명한다. 이 논문에서 사용자 프로파일과 가지형 패턴은 같은 것을 의미한다.

### 4.1 사용자 프로파일의 시퀀스 변환

FiST 시스템은 XPath 식으로 명세된 사용자 프로파일을 Prüfer 시퀀스로 변환한다. 가지형 패턴에는 두 노드 간의 부모-자식('/') 관계와 조상-후손('//') 관계를 포함하고 있다.

먼저 가지형 패턴에 나타나는 노드들을 트리의 노드로 생각하여 가지형 패턴을 트리 구조로 생각한다. 가지형 패턴에서 나타나는 '/'와 '//'는 둘 다 일반적인 트리 간선으로 처리한다. 그림 3의 가지형 패턴 $Q_1$의 트리 표현을 보자. $Q_1$의 LPS는 D B A G E F E A이다. 각각의 사용자 프로파일마다 LPS 레이블 이외에 노드 사이의 관계(부모-자식 또는 조상-후손)와 브랜치 노드 정보와 같은 부가적인 정보를 저장한다. 이런 구조를 프로파일 시퀀스라 한다. 각각의 프로파일 시퀀스는 노드의 순서 리스트로 표현된다. 가지형 패턴에서 부모-자식(또는 조상-후손) 관계를 가지는 노드들의 관계 정보는 해당되는 자식(또는 후손)노드에 저장한다. 프로파일 시퀀스의 각 노드(시퀀스 노드)들은 **Label**, **Qid**, **Pos**와 **Sym**의 4가지 속성을 가진다. 속성 **Label**은 Prüfer 시퀀스 레이블을 저장하고, **Qid**는 유일한 식별자를 나타내고, **Pos**는 프로파일 시퀀스에서 위치를 표현하고,




가지형 패턴 $Q_1$ : /A[B//D]//E[G]/F
$LPS(Q_1)$ = D B A G E F E A

그림 3 가지형 패턴의 시퀀스로의 변환

**Sym**은 표 1에서 나열한 값들의 조합을 가진다. 프로파일 시퀀스의 노드 q가 주어질 때 4가지 속성은 각각 $q_{Label}$, $q_{Qid}$, $q_{Pos}$와 $q_{Sym}$으로 표시할 수 있다.

표 1 심볼 값

| 값 | 설명 |
|---|---|
| '/' | 부모-자식 관계 |
| '//' | 조상-후손 관계 |
| '$' | 브랜치 노드 |
| '#' | 프로파일 시퀀스의 루트 노드 |

**예제 2.** 그림 4(a)는 2개의 가지형 패턴 $Q_1$과 $Q_2$의 프로파일 시퀀스를 보여 주고 있다. $Q_1$의 LPS 는 D B A G E F E A이므로 프로파일 시퀀스는 8개의 노드로 구성된다. $Q_2$의 LPS는 E B C B이므로 프로파일 시퀀스는 4개의 노드로 구성된다. 관계는 프로파일 시퀀스에서 노드의 **Sym** 속성에 저장된다. 예를 들어서 Q1의 프로파일 시퀀스에서 노드 D의 속성 **Sym**은 '//' 값을 가진다. 왜냐하면 첫 번째 노드 D와 두 번째 노드 B가 가지형 패턴 $Q_1$에서 조상-후손 관계를 가지기 때문이다. $Q_1$은 2개의 자식 노드들을 가지고 있는 브랜치 노드가 A와 B 2개 존재한다. 따라서 세 번째와 다섯 번째 노드, 일곱 번째 노드와 여덟 번째 노드들은 **Sym** 속성 값으로 '$'를 가진다. 몇 개의 브랜치 노드들은 동시에 2가지 값을 가질 수 있다. 예를 들어서, $Q_1$의 프로

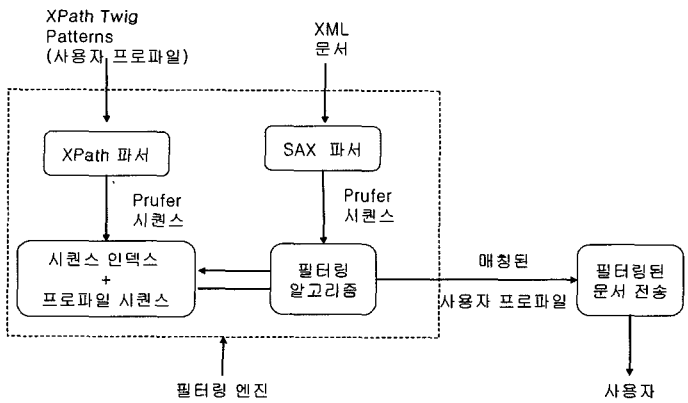



그림 2 FiST의 전체 구조

파일 시퀀스에서 **Label** 속성 값으로 'E'를 가지고 있는 일곱 번째 노드는 **Label** 속성 값 'A'를 가지고 있는 여덟 번째 노드와 조상-후손 관계를 가진다. 따라서 일곱 번째 노드의 **Sym** 속성 값은 2개의 값 '$'와 '//'를 가진다. 프로파일 시퀀스에서 마지막 노드는 가지형 패턴의 루트 노드에 대응한다. 이 노드를 프로파일 시퀀스의 루트 노드라고 부르고 **Sym** 속성 값으로 '#'을 가진다.

### 4.2 사용자 프로파일의 인덱싱

입력 XML 문서에 대하여 매칭 여부를 판단하길 원하는 많은 수의 사용자 프로파일이 주어졌을 때 효율적인 매칭을 위하여 사용자 프로파일을 조직화하는 인덱싱 전략이 필요하다. 이 절에서는 프로파일 시퀀스를 저장하는 인덱스 구조를 설명한다.

개념적으로, FiST 시스템에서 필터링 알고리즘의 첫 번째 단계는 입력 문서와 일치하는 가지형 패턴의 수퍼셋(superset)을 찾기 위해서 입력 문서로부터 생성한 시퀀스와 사용자 프로파일에서 생성한 시퀀스들 사이의 서브시퀀스 매칭 여부를 판단하는 것이다. 정리 1은 가지형 패턴과 XML 문서의 시퀀스 표현사이에 존재하는 관계를 보여준다.

**정리 1([4])** 만약 트리 Q가 트리 T의 서브 그래프이면, LPS(Q)는 LPS(T)의 서브시퀀스 이다.

프로파일 시퀀스의 노드는 그림 4(c)와 같은 상태 기계(state machine)로 매핑할 수 있다. 간략하게 설명하기 위하여 그림 4(c)에서 상태 전이를 많이 생략하였다. 입력 XML 문서의 새로운 태그가 파싱될 때, 상태 기계에서 해당되는 상태 전이를 수행하게 된다. 상태 기계가 최종 상태에 도착하게 되면, 프로파일 시퀀스는 문서 시퀀스의 서브시퀀스 이다. 효율적인 필터링을 위해서는 프로파일 시퀀스들을 동시에 사용하여 입력 문서의 시퀀스와 서브시퀀스 여부인지를 판단해야 한다. 이런 목적을 위해서 해시 기반의 동적 인덱스인 시퀀스 인덱스를 유지한다.

**예제 3.** 그림 4(b)는 시퀀스 인덱스를 보여준다. XML 태그 이름이 해시 테이블의 키로 사용된다. 각각의 키는 프로파일 시퀀스의 노드들을 저장하고 있는 리스트를 가지고 있다. 필터링의 시작 단계에서는 프로파일 시퀀스들의 첫 번째 노드들이 해시 기반의 시퀀스 인덱스에 저장된다. 입력 XML 문서의 새로운 Prüfer 시퀀스 레이블을 얻을 때 마다, 상태 전이가 발생하게 되고 새로이 방문한 상태에 해당되는 노드들이 시퀀스 인덱스에 추가된다. 그림 4(b)의 시퀀스 인덱스는 가지형 패턴 $Q_1$과 $Q_2$의 첫 번째 노드들의 **Label** 속성 값 'D'와 'E'를 해시 키로 사용하여 노드들을 저장하고 있다.

### 4.3 필터링 알고리즘

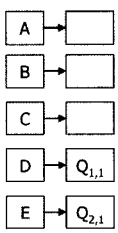
이 절에서는 점진적인 서브시퀀스 매칭 단계와 뒤 따르는 브랜치 노드 검증 단계를 포함하는 필터링 알고리즘을 설명한다. 정리 1은 필요 조건이지 충분 조건이 아니다. 서브시퀀스 매칭 단계에서 필터링 알고리즘은 대부분의 잘못된 결과를 줄이기 위해 필요한 테스트를 수행한다. 프로파일 시퀀스 노드 q가 주어질 때, 테스트는 $q_{Sym}$의 값을 이용하여 진행된다. 이 테스트를 용이하게 하기 위해서 필터링 알고리즘은 하나의 실행시간 스택

$Q_1$ : /A[B//D]//E[G]/F  LPS($Q_1$) = D B A G E F E A

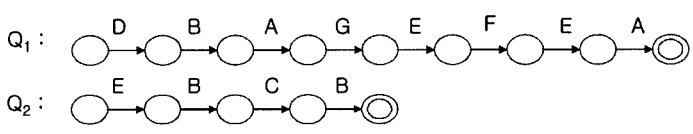
| Label | D | B | A | G | E | F | E | A |
|---|---|---|---|---|---|---|---|---|
| Qid | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Seq | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Sym | // | / | $ | / | $ | / | $// | $# |

$Q_2$: //B[E]/C  LPS($Q_2$) = E B C B

| E | B | C | B |
|---|---|---|---|
| 2 | 2 | 2 | 2 |
| 1 | 2 | 3 | 4 |
| // | $ | / | $# |




(a) 프로파일 시퀀스 (b) 시퀀스 인덱스




(c) 단순화된 상태 기계

그림 4 프로파일 시퀀스, 시퀀스 인덱스와 상태 기계

(runtime stack)을 이용한다. 실행시간 스택은 현재 처리 되고 있는 태그(엘리먼트)에서부터 문서의 루트에 이르는 경로들에 있는 태그들을 저장하고 있다. 문서의 엘리먼트들은 문서 방문 순서에 따라 스택에 푸시(push)되고 팝(pop)된다. 스택의 최대 깊이는 입력 문서의 최대 높이를 넘지 않는다.

4.3.1 점진적인 서브시퀀스 매칭(progressive subsequence matching)

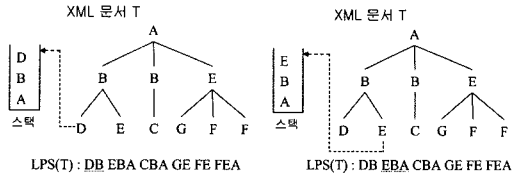
서브시퀀스 매칭이 점진적이라 불리는 이유는, 문서의 시퀀스 표현을 점진적으로(incrementally) 생성하고 이때 생성된 문서 시퀀스의 서브시퀀스에 해당되는 프로파일 시퀀스를 찾기 때문이다. 문서가 파싱될 때 실행시간 스택의 태그들을 조사함으로써 XML 문서의 LPS를 구성할 수 있다. FiST 시스템의 SAX 파서가 입력 XML 문서를 파싱한다. 필터링 알고리즘을 수행하기 위해서 SAX 파서의 StartTagHandler와 EndTagHandler 프로시저에 수정이 필요하다. 알고리즘 1은 StartTagHandler와 EndTagHandler 프로시저를 보여준다. 태그 이름과 함께 StartTagHandler가 호출되면, 태그 이름은 1행과 같이 스택에 푸시된다. EndTagHandler가 호출되면 LPS를 생성하기 위해서 엘리먼트 태그가 문서에서 리프 노드인지 확인한다(2 행). 만약 태그가 리프 노드이면, 스택의 최상위 엘리먼트가 다음 Prüfer 시퀀스 레이블이 되고 필터링 프로시저인 FindSubsequence 함수를 호출한다(3 행). 태그가 리프 노드이던 아니던, 스택의 최상위 원소를 제거한다. 새로운 최상위 엘리먼트가 다음 Purfer 시퀀스 레이블이 되고(4 행), 필터링 함수를 다시 호출한다(5행).

```
    stack S;                                /* 실행시간 스택 */
    procedure StartTagHandler(tag)
1       S.push(tag);
    end

    procedure EndTagHandler(tag)   /* Prüfer 시퀀스 사용 */
2       if tag is a leaf node then
3           FindSubsequence(S.top());
        end
4       S.pop();
5       FindSubsequence(S.top());
    end
```

알고리즘 1. SAX Handlers

EndTagHander가 호출될 때마다, 스택의 최상위 원소는 문서를 위한 LPS의 i번째 레이블이 된다. 필터링 알고리즘은 시퀀스 인덱스를 사용하여 모든 프로파일 시퀀스에 대하여 서브시퀀스를 동시에 구하게 된다. 그 결과, 문서와 문서의 LPS는 한 번만 스캔하게 된다. 전역 스택을 접근하여 LPS를 점진적으로 생성할 수 있기 때문에 문서에서 생성된 전체 시퀀스를 실제적으로 저장할 필요는 없다.




(a) EventHandler of D (b) EventHandler of E

그림 5 LPS(T)의 생성 과정

**예제 4.** 그림 5는 XML 문서 트리 T에서 LPS를 구성하는 과정을 설명한다. 이 문서의 LPS(T)는 "D B E B A C B A G E F E F E A"이다. StartTagHandler가 호출될 때 마다, 엘리먼트들은 스택에 푸시된다. 엘리먼트 D의 EndTagHandler가 불릴 때 전역 스택의 내용은 그림 5(a)와 같다. 엘리먼트 D는 XML 문서 트리 T에서 리프 노드이다. 스택의 최상위 원소가 LPS(T)의 첫 번째 레이블이 된다. 스택의 최상위 원소 D를 제거한다. 새로운 최상위 원소 B가 LPS(T)의 두 번째 레이블이 된다. B는 여전히 스택에서 유지되고 있다. B의 자식인 E의 EndTagHandler가 호출될 때 스택의 내용은 그림 5(b)와 같다. E는 T의 리프 노드이므로 스택의 최상위 원소 E가 LPS(T)의 세 번째 레이블이 된다. 그런 다음 최상위 원소 E를 제거한다. 스택의 새로운 최상위 원소 B가 LPS(T)의 네 번째 레이블이 된다. T의 엘리먼트 B의 EndTagHandler가 호출될 때, 스택에는 B와 A가 존재한다. 엘리먼트 B는 XML 문서 트리 T에서 리프 노드가 아니므로, 최상위 원소 B를 스택에서 제거한다. 스택의 새로운 최상위 원소 A가 LPS(T)의 다섯 번째 레이블이 된다. XML 문서 트리 T의 루트 엘리먼트(A)의 EndTagHander가 호출될 때 까지 이 과정을 반복한다

필터링 과정의 핵심 동작은 알고리즘 2와 같다. FindSubsequence 프로시저는 EndTagHandler에서 호출된다. 테스트할 노드를 찾기 위해서 레이블 L을 키로 사용하여 시퀀스 인덱스를 검색한다(1 행). 리스트에 있는 모든 노드 q마다 $q_{Sym}$의 값에 따라서 해당되는 동작을 수행한다(6-11 행). $q_{Sym}$이 값들의 리스트이기 때문에, 4행에 있는 동작을 반복 수행한다.

**4.3.2 실행시간 스택 사용의 이점**

서브시퀀스 매칭 과정 중에, FiST 시스템은 실행 시간 스택을 사용하여 대부분의 잘못된 매치를 걸러낸다. 태그 이름이 매치되고 알고리즘 3의 스택 검사도 성공해야 상태 전이(그림 4(c))가 수행된다. 실행 시간 스택을 이용하면 부모-자식 관계와 조상-후손 관계의 판단

```
   Input: Prüfer 시퀀스 레이블 L
   procedure FindSubsequence(L)
1      CurrentList ← SequenceIndex[L];
2      foreach SequenceNode q in CurrentList do
3          test ← false;
4          foreach value v in qSym do
5              switch v do
6                  case '/' or '//' :   /* 부모-자식 또는 조상-후손 관계 */
7                      if doSimpleStackTest(q,v) = true then test ← true;
8                  case '$':           /* 브랜치 노드 */
9                      doBranch(q);
10                 case '#':           /* 루트 노드 */
11                     BranchNodeVerification(qQid);
               end
           end
12         if ((qSym = '/' or qSym = '//') and ( test = true)) or (qSym = '$') then
13             q' ← NextNode(q);
14             copy q' into Sequence Index using key q'Label;
       end
   end
   end
```

알고리즘 2. 점진적인 서브시퀀스 매칭

을 쉽게 할 수 있고, 시퀀스 인덱스에 불필요한 노드 복사를 피할 수 있고, 서브시퀀스 매칭의 범위를 제한하여 효율적인 필터링에 도움이 된다. 이 절에서는 이런 장점에 대해 자세히 설명한다.

```
   Input: 프로파일 시퀀스 노드 q
          q의 속성 sym의 값 v
   procedure doSimpleStackTest(q, v)
1      q' ← NextNode(q);
2      if (v = '/' and TestPC(q, q') is true) OR
         (v = '//' and TestAD(q, q') is true) then
3          return true;
4      else return false;
   end
```

알고리즘 3. 스택 검사

**(a) 부모-자식 관계와 조상-후손 관계의 처리**

실행시간 스택은 프로파일 시퀀스의 노드와 매치하는 입력 문서의 노드들 사이에서 부모-자식 관계와 조상-후손 관계를 테스트할 때 사용된다. 이 두 테스트를 TestPC와 TestAD라 하자. 프로파일 시퀀스에 있는 두 개의 노드 q와 q′(q′← NextNode(q))를 고려하자. 스택에서 $q_{Label}$의 바로 아래에 $q'_{Label}$이 존재한다는 TestPC(q, q′)의 결과는 참이다. 반면에 스택에서 $q_{Label}$의 아래쪽에 $q'_{Label}$이 존재하기만 하면 TestAD(q, q′)의 결과는 참이 된다.

예를 들어, 그림 6은 E의 EndTagHandler가 호출될 때의 스택의 상태를 보여준다. $Q_i$의 프로파일 시퀀스에서 j번째 엘리먼트는 스택의 최상위 원소와 매치된다. j번째 엘리먼트는 '//'를 sym 값으로 가지고 있으므로,

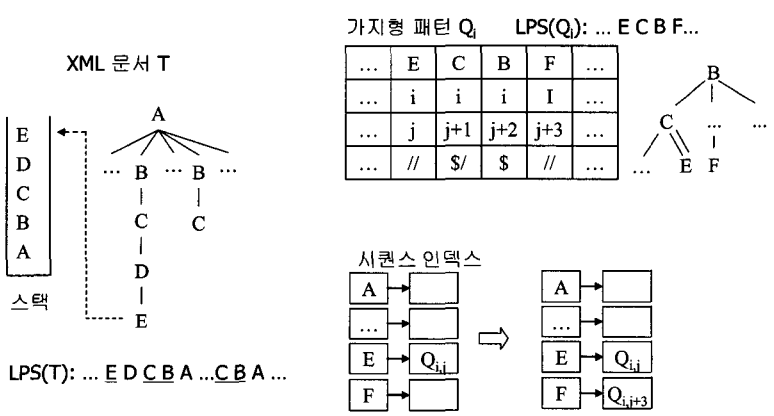



그림 6 실행시간 스택 사용의 이점

TestAD($Q_{i,j}$, $Q_{i,j+1}$)을 이용한다. 스택에서 엘리먼트 C는 엘리먼트 E의 아래에 존재하고 있으므로 TestAD의 결과는 참이다. 이 테스트는 가지형 패턴에서 두 노드 $Q_{i,j+1}$과 $Q_{i,j}$ 사이에 존재하는 조상-후손 관계가 문서 T에서도 만족되고 있음을 의미한다. 다음으로 j+1번째 노드 $Q_{i,j+1}$이 사용된다. $Q_{i,j+1}$는 스택에서 엘리먼트 C와 매칭된다. 또한 엘리먼트 B는 스택에서 C의 바로 밑에 존재하고 있다. 이것은 두개의 노드 $Q_{i,j+2}$와 $Q_{i,j+1}$이 부모-자식 관계를 만족하고 있음을 의미하고, TestPC의 결과는 참이라는 뜻이다. 프로시저 위의 동작을 수행하기 위해서 FindSubsequence에서 doSimpleStackTest 함수를 호출한다(6 행).

그림 6의 가지형 패턴은 입력 문서의 어느 곳에나 매칭될 수 있다. 그러나 j+2번째 노드인 $Q_{i,j+2}$가 문서의 루트('/')와 매치되어야 한다면, 실행 시간 스택에서 $Q_{i,j+2}$에 대응되는 엘리먼트인 B가 스택의 최상위 원소인지 확인하면 된다.

### (b) 시퀀스 인덱스로의 불필요한 노드 복사 피함

그림 6의 가지형 패턴 $Q_i$를 다시 고려하자. 알고리즘 1을 사용하면, EndTagHandler가 호출될 때마다 FindSubsequence 함수가 호출된다. 이 예제에서, 리프 엘리먼트 E의 EndTagHandler가 호출될 때마다, 스택에 있는 엘리먼트들은 문서 LPS(T)의 일부분인 "E D C B A"를 표현하고 있다. 여기서 엘리먼트 A는 브랜치 노드이다. 단순하게 처리를 한다면 알고리즘 1을 사용하여 "E D C B A" 순서로 매번 FindSubsequence 함수를 호출하면 된다. 이 방법은 FindSubsequence 함수가 프로파일 시퀀스에 있는 노드와 매치될 때마다 프로파일 시퀀스의 다음번 노드가 시퀀스 인덱스로 복사되는 것을 필요로 한다. 그렇지만 실행 시간 스택은 XML 문서에서 브랜치 노드 A까지 이르는 경로의 모든 LPS의 레이블인 "E D C B A"를 저장하고 있으므로, 실행 시간 스택을 미리 보기(look-ahead) 버퍼로 사용할 수 있다. 따라서 doSimpleStackTest를 매번 수행하지 않고, 재귀적인 스택 테스트를 사용하여 프로파일 시퀀스에서 브랜치 노드까지에 있는 노드들을 시퀀스 인덱스에 복사하는 불필요한 동작을 피할 수 있다.

알고리즘 4는 이런 동작을 설명한다. 알고리즘 4에서 스택 검사(2 행)의 결과가 참일 때마다, 프로파일 시퀀스 q의 다음번 노드 q'를 사용하여 브랜치 노드가 나타날 때까지 스택 테스트를 반복 실행한다. 재귀 스택 테스트가 성공적으로 종료되면, 프로파일 시퀀스에서 브랜치 노드의 다음 노드가 시퀀스 인덱스로 복사된다. 이 때 리프 노드에서 브랜치 노드까지의 경로에서 나타나는 중간 노드들이 시퀀스 인덱스에 복사되는 것을 방지할 수 있다. 여기서 브랜치 노드는 다음 노드와 아무런 관계를 가지지 않는 브랜치 노드를 말한다. 이 때 브랜치 노드의 다음 노드는 새로운 경로의 리프 노드이다. 브랜치 노드 중에서 다음 노드와 부모-자식이나 조상-후손 관계를 가지는 노드들은 doRecursiveStackTest를 다음 노드와 아무런 관계를 가지지 않는 브랜치 노드가 나타날 때까지 계속해서 수행하게 된다. 알고리즘 2에서 doSimpleStackTest(6 행)을 doRecursiveStackTest로

```
    Input: 프로파일 시퀀스 노드 q
           q의 속성 sym의 값 v
    procedure doRecursiveStackTest(q, v)
1       q' ← NextNode(q);
2       if (v = '/' and TestPC(q, q') is true) OR
          (v = '//' and TestAD(q, q') is true) then
3           foreach value v' in q'Sym do
4               switch v' do
5                   case '/' or '//' :   /* 부모-자식 또는 조상-후손 관계 */
6                       doRecursiveStackTest(q',v');
7                   case '$':           /* 브랜치 노드 */
8                       doBranch(q);
9                   case '#':           /* 루트 노드 */
10                      BranchNodeVerification(q'Qid);
                end
            end
11          if qSym = '$' then
12              copy q' into Sequence Index using key q'Label;
            end
        end
    end
```

알고리즘 4. 재귀 스택 검사

대체하고, 10-12 행을 생략하면 재귀 스택 테스트를 사용하는 FindSubsequence 함수가 된다.

**(c) 서브 시퀀스 매칭의 범위 제한**

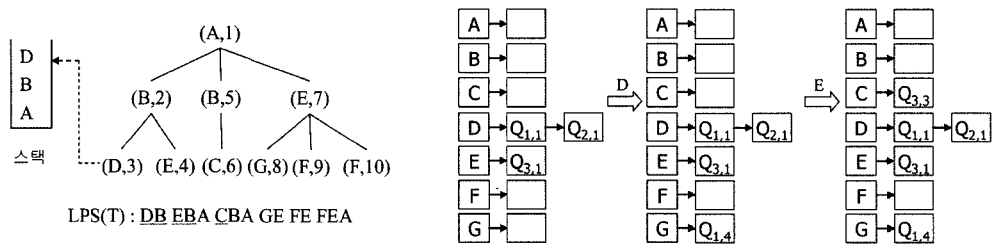
실행 시간 스택의 또 다른 중요한 이점은 서브시퀀스 매칭에서 문서 시퀀스의 범위를 제한한다는 것이다. 그림 6의 XML 문서 T와 가지형 패턴 $Q_i$를 보자. 문서 T의 LPS(T)는 "... E B C B A ... C B A..."이고, 가지형 패턴 $Q_i$의 LPS($Q_i$)는 "... E C B ..."이다. LPS(T)에서 "E C B"와 매칭하는 서브시퀀스가 2개 존재한다(그림 6에서 밑줄 친 부분). XML 문서에서, 뒤에 나타나는 엘리먼트 C(T에서 리프 노드)와 그 노드의 부모 노드인 B는 엘리먼트 E와 관계를 가지고 있지 않다. Prüfer 시퀀스 레이블 E가 생성될 때, 스택에는 오직 하나의 C와 B만 존재한다. 따라서 필터링 알고리즘은 오직 하나의 서브시퀀스 인스턴스 "E C B"에 대해서 검사를 수행한다. 그 결과 스택은 실제 매치를 표현하지 않는 서브시퀀스에 대한 계산을 가지치기(pruning)하는 능력을 제공한다.

4.3.3 XML 문서 필터링 예제

아래의 예제 5는 그림 7(a)의 XML 문서 T와 그림 7(c)의 가지형 패턴 $Q_1$, $Q_2$와 $Q_3$를 이용하여 필터링 알고리즘의 수행 과정을 설명한다.

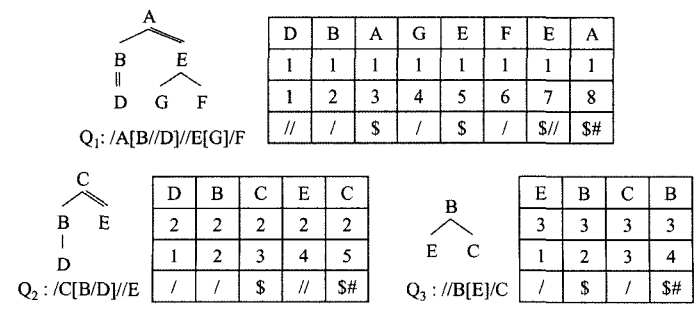
**예제 5.** 그림 7(a)는 FindSubsequence(D)가 호출될 때의 실행시간 스택을 보여준다. 시퀀스 인덱스의 키 값 D에 존재하는 노드 리스트가 먼저 처리된다. 두 개의 노드 $Q_{1,1}$ 과 $Q_{2,1}$이 존재한다. 먼저 $Q_{1,1}$의 처리 과정을 보자. $Q_{1,1}$의 다음 노드는 $Q_{1,2}$이고, $Q_{1,2_{Label}} = B$이고 $Q_{1,1_{Sym}} = '//'$인데 실행 시간 스택의 원소 D 밑에 원소 B가 존재하므로 TestAD의 결과는 참이다. 다음으로 doRecursiveStack($Q_{1,2}$, '//')이 수행된다. $Q_{1,2}$의 다음 노드는 $Q_{1,3}$이다. 스택에서 원소 A는 원소 B의 바로 아래에 존재하므로 TestPC의 결과는 참이다. $Q_{1,3}$은 브랜치 노드이고 다음 노드 $Q_{1,4}$와 아무런 관계를 가지지 않으므로, 다음 노드 $Q_{1,4}$가 시퀀스 인덱스의 키 값 G($Q_{1,4}$의 레이블)의 노드 리스트에 추가된다. 다음으로 $Q_{2,1}$의 처리 과정을 보자. $Q_{2,2}$와 $Q_{2,1}$을 이용한 TestPC의 결과는 참이므로 doRecursiveStack($Q_{2,2}$, '//')을 호출한다. 그러나 $Q_{2,3}$의 레이블 해당되는 스택의 원소 C가 현재 스택에 존재하지 않으므로, $Q_{2,3}$와 $Q_{2,2}$을 이용한 TestPC의 결과는 거짓이다. 이 때 어떠한 노드 복사도 행해지지 않으며, 그림 7(b)와 같이 시퀀스 인덱스에는 $Q_{2,1}$만 그대로 존재하게 된다.

다음으로 FindSubsequence(B)가 호출된다. 시퀀스 인덱스의 키 값 B에는 어떠한 노드도 존재하지 않기 때문에, 어떤 동작도 수행하지 않는다. 다음으로 FindSubsequence(E)가 호출되고, $Q_{3,1}$과 $Q_{3,2}$의 스택 테스트의 결과는 참이다. $Q_{3,2}$가 브랜치 노드이므로 다음 노드 $Q_{3,3}$가 시퀀스 인덱스의 키 값 C의 리스트에 복사된다.




(a) XML 문서 트리 T

(b) 시퀀스 인덱스의 변화




(c) 가지형 패턴들

그림 7 서브시퀀스 매칭 예제

다음으로 FindSubsequence(B)가 다시 호출되고, 아무런 동작도 수행하지 않는다. FindSubsequence(C)가 호출될 때, 스택에는 원소 C B A가 존재한다. 이때 시퀀스 인덱스의 키 값 C의 노드 리스트에는 오직 $Q_{3,3}$만 존재한다. $Q_{3,3_{gm}}$='/'이고 스택에서 원소 B가 원소 C 아래에 존재하므로, $Q_{3,4}$와 $Q_{3,3}$의 스택 테스트의 결과는 참이다. $Q_{3,4}$는 브랜치 노드이자 루트 노드인데, 이 노드의 처리 방법은 4.3.4절에서 설명한다.

4.3.4 브랜치 노드와 루트 노드의 처리

서브시퀀스 매칭만을 이용한 필터링은 잘못된 결과들도 포함하고 있다[4]. FiST 시스템에서 이러한 잘못된 결과들을 제거하기 위해서, 가지형 패턴에서 브랜치 노드의 결합성(connectivity)을 검증하는 단계를 사용한다. 편리성을 위하여 각각의 입력 XML 문서에서 태그들은 전위 순회 번호를 가지고 있다고 가정한다.[1] XML 문서를 SAX 파싱을 할 때, 엘리먼트에 전위순회 번호를 부여하기 위해서 counter 변수를 사용하다. StartTag-Handler가 호출될 때 마다 counter 변수를 증가시키고 이 변수의 값을 전위순회 번호로 사용한다.

**예제 6.** 그림 7의 예제를 보자. XML 문서 트리 T의 엘리먼트 옆의 숫자들은 전위순회 번호를 의미한다. LPS(T) = D B E B A C B A G E F E F E A이다. 그림 7(b)에서 두 개의 가지형 패턴 $Q_1$과 $Q_3$를 보자. LPS($Q_1$) = D B A G E F E A이고, LPS($Q_3$) = E B C B이다. LPS($Q_1$)과 LPS($Q_3$)는 LPS(T)의 서브시퀀스임을 알 수 있다. 이것은 $Q_1$과 $Q_3$가 문서 T와 매칭하는 가지형 패턴 후보들임을 의미한다. 그러나 $Q_3$는 실제로는 문서 T와 매치가 아니다. 왜냐하면 문서 T에서 엘리먼트 E와 엘리먼트 C를 동시에 자식으로 가지고 있는 엘리먼트 B가 존재하지 않기 때문이다. $Q_3$는 두 개의 서로 다른 엘리먼트 B 즉 (B,2)와 (B,5)에 매칭된다. 이러한 잘못된 매치를 제거하기 위해서 서브시퀀스 매칭을 수행할 때, 프로파일 시퀀스에 있는 노드 B가 문서 T에서 하나의 노드에만 매칭되는지 여부를 확인해야 한다. 반면에 LPS($Q_1$)에서 두 개의 노드 E는 문서 트리 T에서 하나의 엘리먼트를 표현하는 두 개의 E와 매칭된다. 또한 LPS($Q_1$)의 두 개의 노드 A는 문서 트리 T에서 하나의 엘리먼트를 표현하는 두 개의 A와 매칭된다. 따라서 $Q_1$는 문서 T와 실제 매치이다.

FiST 시스템은 잘못된 매치를 버리는 검증 단계를 효율적으로 수행하기 위해서 프로파일 시퀀스에 나타나는 브랜치 노드 위해서 특별한 동작을 수행한다. 브랜치 노드 처리의 핵심은 브랜치 노드와 매칭되는 입력 XML 문서의 태그(엘리먼트)들의 정보를 저장하는 것이다. 이런 정보들이 결합성을 확인하는 검증 단계에서 사용된다. 시퀀스 노드와 매칭된 문서의 노드가 가지는 전위순회 번호를 기록하기 위해서 BranchID Set라 불리는 자료 구조가 사용된다.

서브시퀀스 매칭 단계에서, 프로파일 시퀀스 노드 q가 브랜치 노드이면 doBranch(q) 프로시저는 매칭된 XML 엘리먼트의 전위순회 번호를 q를 위한 BranchID set에 저장한다. Prüfer 시퀀스에서, 주어진 노드가 나타날 회수는 그 노드의 자식 노드의 수와 동일하다[4]. 따라서 프로파일 시퀀스에서는 두 종류의 브랜치 노드가 존재한다. 예를 들어, 그림 7(c)에서 프로파일 시퀀스 노드 $Q_{1,5}$와 $Q_{1,7}$는 가지형 패턴에서 브랜치 노드 E에 해당된다. $Q_{1,5}$는 브랜치 노드 E의 마지막 출현이 아니므로, 내부 브랜치 노드(internal branch node)라 한다. 내부 브랜치 노드는 프로파일 시퀀스에서 다음 노드와 아무런 관계를 가지지 않는다. 따라서 필터링 알고리즘에서 어떠한 스택 테스트도 수행하지 않고, 다음 노드를 시퀀스 인덱스에 복사한다. 그러나 마지막에 나타나는 브랜치 노드는 프로파일 시퀀스에서 다음 노드와 부모-자식 관계나 조상-후손 관계를 가진다. 예를 들어 그림 7(c)의 $Q_{1,7}$은 $Q_{1,8}$과 조상-후손 관계를 가진다. 이러한 노드들을 마지막 브랜치 노드(final branch node)라 한다. 마지막 브랜치 노드는 BranchID set에 매칭된 엘리먼트의 전위순회 번호를 저장하는 것 이외에 스택 테스트도 수행해야 한다. doRecursiveStackTest에서 재귀 호출은 내부 브랜치 호출에서 종료된다.

그림 7(c)에서 브랜치 노드 A에 해당되는 $Q_{1,3}$과 $Q_{1,8}$을 위한 두 개의 BranchID set이 필요하고, 브랜치 노드 E에 해당되는 $Q_{1,5}$와 $Q_{1,7}$을 위한 두 개의 BranchID set이 필요하다.

가지형 패턴의 루트 노드에 도착했다는 것은 그 가지형 패턴이 입력 XML 문서의 서브시퀀스라는 의미이다. 매칭된 서브시퀀스가 입력 XML 문서에 실제 가지형 패턴인지를 검사하기 위한 브랜치 노드 검증 단계(Refinement by Branch Node Verification)을 수행한다. 각각의 프로파일 시퀀스마다, 서브시퀀스 매칭 단계가 수행될 때 BranchID set들이 구성된다. 프로파일 시퀀스의 루트 노드를 만날 때 마다, 알고리즘 2나 알고리즘 4에서 프로시저 BranchNodeVerification을 호출한다. 후보 가지형 패턴에서 각각의 브랜치 노드마다, BranchID set들의 교집합을 계산하고, 그 결과가 공집합이 아니라는 것은 가지형 패턴의 브랜치 노드 XML 문서에서 실제 연결이 되었다는 것을 의미한다. 만약 가지형 패턴에 존재하는 모든 브랜치 노드들이 모두 연결

1) 후위순회 번호와 같은 번호 부여 기법(numbering scheme)도 가능하다. 그렇지만 문서의 태그들은 문서 방문 순서대로 파싱되므로 전위순회 번호가 적당하다고 볼 수 있다.

되었다면, 이 가지형 패턴은 최종 결과가 된다.

예를 들어, 그림 7(c)에 있는 가지형 패턴 $Q_1$를 고려하자. 노드 A를 위한 BranchID set의 교집합의 결과는 {1}이고, 노드 E를 위한 교집합의 결과는 {7}이다. 두 개의 브랜치 노드 각각의 교집합 결과가 공집합이 아니므로, $Q_1$은 입력 문서 T의 매칭이다.

## 5. 실 험

### 5.1 실험 환경과 데이타셋

이 논문에서는 FiST 시스템과 YFilter 시스템을 비교하였다. 실험은 512MB의 메모리를 가지고 있고 리눅스가 설치되어 있는 *Intel Pentium IV 2.4 GHz CPU* 기계에서 수행하였다. FiST 시스템은 Xerces XML Parser 2.5.0 버전[15]을 사용하여 c++로 구현되었으며, GNU g++ 3.3.2를 사용하여 컴파일 하였다. YFilter 시스템은 Java로 구현되었으며 Java 가상 머신 1.4.2.를 지니고 있는 Eclipse 3.0.1 버전에서 컴파일하고 수행하였다.

실험을 위하여 TreeBank DTD를 사용하여 생성한 XML 문서를 사용하였다. XML Generator[16]를 사용하여 최대 깊이가 36인 XML 문서들을 1000개 생성하였다. 생성된 문서들은 그 크기에 따라서 [1KB, 10KB), [10KB, 20KB), [20KB, 30KB]와 [30KB, 123KB)로 분류하였다. 이 논문에서는 편의상 이 데이타셋을 1k, 10k, 20k, 30k라 한다.

YFilter의 XPath Generator를 사용하여 균등 분포 프로파일들을 생성하였다. 균등 분포는 엘리먼트 이름들이 균등 분포(uniform distribution)를 가지는 데이타셋을 의미한다. 가지형 패턴의 최대 깊이는 10으로 고정하였다. 각각의 데이타셋에서 가지형 패턴이 가질 수 있는 브랜치의 개수를 3에서 7까지 변화시켰다. 또한 데이타셋이 가질 수 있는 사용자 프로파일의 개수도 50,000개부터 150,000개까지 25,000개 단위로 변화시켰다.

### 5.2 성능 척도

YFilter와 FiST를 다양한 확장성 측면에서 필터링 비용의 경향을 관찰하여 비교하였다. 왜냐하면 YFilter는 Java로 구현되었고 FiST는 C++로 구현되었기 때문이다. 서로 다른 언어로 구현된 2개의 시스템의 공평한 비교를 위하여, YFilter와 FiST의 성능을 scaleup이라는 척도와 실행 시간(wall clock time)을 비교하였다. 실행 시간을 측정할 때, 주어진 가지형 패턴 집합과 데이타셋에 대하여 문서마다 평균 필터링 시간을 측정하였다. 필터링 시간은 문서 파싱 시간과 필터링 알고리즘이 사용하는 시간을 포함한다. 파싱 타임은 필터링 시간에 비하여 매우 작은 값이다. 예를 들어, YFilter가 데이타셋 30k를 파싱하는데 걸리는 평균 시간은 30 ms이다.

scaleup 성능을 측정하기 위하여, 다음과 같은 수식을 사용하였다.

$$scaleup = \frac{tAvg - tAvg_{base}}{tAvg_{base}}$$

tAvg는 실험에서 측정한 필터링 시간이고 $tAvg_{base}$는 기본 상태(base case)를 위한 필터링 시간이다. 측정할 확장성의 종류에 따라서, $tAvg_{base}$는 가장 작은 수의 가지형 패턴에 대한 필터링 시간이거나, 가장 작은 수의 브랜치를 가지는 가지형 패턴에 대한 필터링 시간이거나, 가장 작은 크기의 입력 문서에 대한 필터링 시간이다. 예를 들어, 가지형 패턴의 수를 변화시키면서 확장성을 측정할 때 $tAvg_{base}$는 50,000개의 사용자 프로파일에 대한 필터링 시간이다. 따라서 scaleup 측정값이 양수(또는 음수)라는 것은 테스트 케이스가 증가함에 따라서 필터링 시간도 증가(또는 감소)함을 의미한다. 다양한 환경에서 실험을 통하여 FiST가 YFilter보다 좋은 확장성(필터링 시간이 더 늦게 증가함)을 가지는 것을 확인하였다.

### 5.3 성능 분석

이 절에서는 FiST와 YFilter의 성능을 가지형 패턴의 크기, 브랜치의 크기와 문서 크기에 대하여 scaelup과 실행 시간으로 분석하였다. FiST는 순서 가지형 패턴 매칭을 지원하고 YFilter는 무선서 가지형 패턴 매칭을 지원하지만, YFilter에 후처리 과정을 더하면 순서 매칭을 수행할 수 있다. YFilter에서 후처리 과정의 비용을 제외하고 시간을 측정하였다.

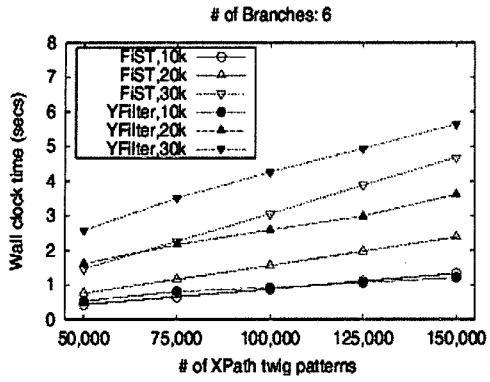
#### 5.3.1 가지형 패턴의 수 변화

그림 8은 가지형 패턴마다 6개와 7개의 브랜치를 가지는 데이타셋 대한 FiST와 YFilter의 실행 시간을 보여준다. 가지형 패턴의 수는 50,000개부터 150,000개까지 25,000 단위로 변화시켰다. 1k에 대한 결과는 10k에 대한 결과와 비슷하기 때문에 생략하였다.

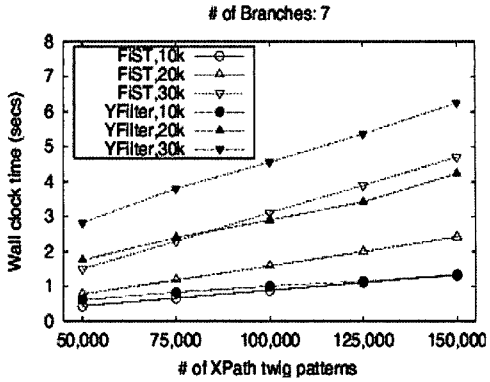
그림 8(a)에서 가지형 패턴에서 브랜치의 수는 6이다. 가지형 패턴이 증가함에 따라 YFilter와 FiST의 필터링 시간도 증가한다. 데이타셋 20k와 30k를 보면, FiST의 확장성이 YFilter보다 훨씬 좋아짐을 알 수 있다. 예를 들어서 150,000개의 가지형 패턴에 대하여 데이타셋 20k 크기의 입력 문서가 주어질 때, FiST는 YFilter보다 34% 빠른 처리 시간을 보였다. 가지형 패턴 마다 7개의 브랜치를 가지고 있는 150,000개의 가지형 패턴(그림 8(b))에 대하여 데이타셋 20k에 해당하는 입력 문서들을 필터링하는 경우 FiST가 YFilter보다 43% 빠른 처리 시간을 보였다. 데이타셋 10k와 1k에 대하여서는 FiST는 YFilter에 필적한 만한 성능을 보여주었다.

#### 5.3.2 브랜치 수 변화

다음으로 가지형 패턴에서 브랜치의 수를 다양하게

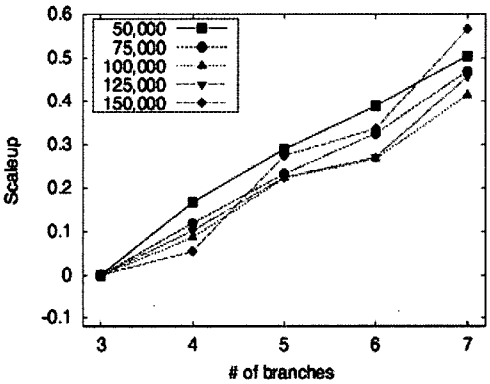



(a) 브랜치의 수 6

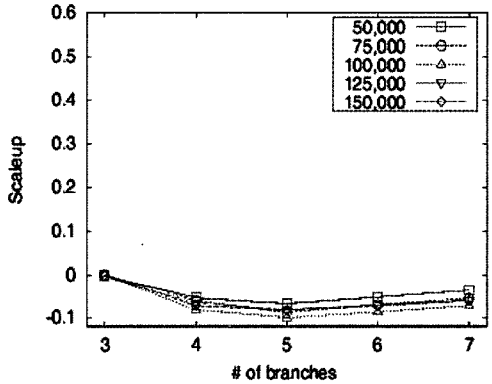



(b) 브랜치의 수 7

그림 8 가지형 패턴(사용자 프로파일)의 개수 변화에 따른 실험 결과




(a) YFilter(20k)




(b) FiST(20k)

그림 9 브랜치 수의 변화에 따른 실험 결과

변화시켜 가면서 FiST와 YFilter의 확장성을 비교하였다. 그림 9는 이 실험에 대한 결과를 보여준다. 그림 9(a)와 (b)는 각각 데이타셋 20k를 사용하였을 때의 YFilter와 FiST의 scaleup을 보여준다. 가지형 패턴에서 브랜치의 수가 3에서 7로 증가함에 따라 YFilter의 필터링 비용도 증가한다. 사용한 모든 데이타셋에서 이런 경향을 발견할 수 있다. 이에 반하여, 브랜치의 수가 증가하여도 FiST의 필터링 비용은 감소한다. 이것은 그림 9(b)의 음수 scaleup 값으로 확인할 수 있다. 이런 결과는 FiST의 전체 가지형 패턴 매칭이 YFilter보다 좋은 확장성을 가짐을 보여준다.

그림 10은 데이타셋 20k에 대한 FiST와 YFilter의 실행 시간을 보여준다. FiST는 브랜치 수가 4,5,6,7일 때 YFilter보다 더 우수한 성능을 가지고 있음을 보여준다. 위에서 언급한 대로 YFilter의 필터링 비용은 증가하고 경향을 보이고 FiST의 필터링 비용은 감소하는 경향을 보이는 것을 관찰할 수 있다. 또한 문서의 크기

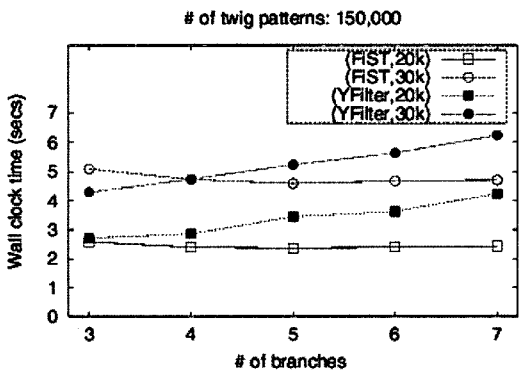



그림 10 필터링 시간

가 증가함에 따라 필터링 비용도 증가하는 것을 알 수 있다.

5.3.3 XML 문서의 크기 변화

이 절에서는 입력 문서의 크기를 증가시키면서 FiST

와 YFilter의 성능을 분석하였다.

그림 11에서 브랜치 수가 4와 6인 경우는 브랜치 수가 3,5,7인 경우와 비슷한 경향을 보이기 때문에 생략되었다. 그림 11은 150,000개의 가지형 패턴들을 엘리먼트의 균등 분포를 이용하여 생성한 데이타셋에 대한 YFilter와 FiST의 scaleup을 보여준다. x축은 입력 문서의 크기를 나타내는 수치이다. YFilter의 scaleup이 FiST의 scaleup보다 빨리 커진다는 것은 YFilter의 필터링 비용이 FiST의 필터링 비용보다 빨리 증가한다는 것을 나타낸다. 문서의 크기가 증가함에 따라 YFilter와 FiST의 차이가 더 벌어짐을 알 수 있다. 이 결과들은 문서 크기가 증가함에 따라서 FiST가 YFilter보다 더 좋은 확장성을 가짐을 보여준다. 그래프에서 3,5,7개의 브랜치를 가질 때의 FiST의 성능 경향이 서로 겹친다는 것은 브랜치의 수가 증가하는 것에도 불구하고 경향은 비슷하다는 것을 나타낸다. 이것은 그림 10에서 보았던 성능 경향과 일치한다.

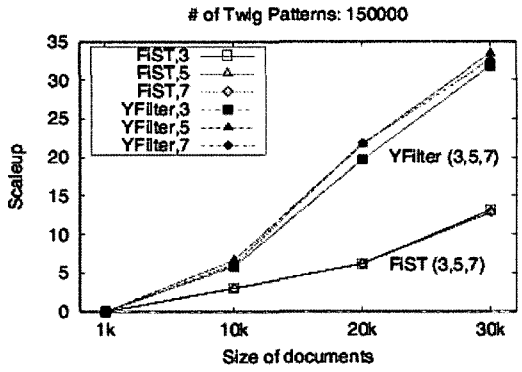



그림 11 XML 문서 크기 변화에 따른 실험 결과

## 6. 결 론

이 논문에서는 XML 문서 필터링 시스템 FiST를 제안하였다. 이전의 연구들은 가지형 패턴을 여러 개의 선형 경로들로 분해하여 각각의 선형 경로에 대하여 매칭을 수행하고 그 결과를 합쳐는 반면에, FiST 시스템은 입력 XML 문서에 대하여 전체 가지형 패턴 매칭을 수행할 수 있다. 이를 위해서 FiST는 XML 가지형 패턴(사용자 프로파일)과 XML 문서들을 Prüfer 시퀀스로 변환한다. 이 논문에서는 Prüfer 시퀀스를 사용하여 점진적 서브시퀀스 매칭을 수행하는 단계와 브랜치 검증을 통하여 잘못된 결과들을 버리는 정제 단계로 구성된 필터링 알고리즘을 제안하였다. 또한 다양한 환경에서 FiST 시스템이 최신의 연구인 YFilter 시스템보다 좋은 결과를 나타냄을 실험으로서 보였다.

## 참 고 문 헌

[1] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie and Jrme Simon, XML Path Language (XPath) 2.0 W3C Working Draft 16. Technical Report WD-xpath-20-20020816, World Wide Web Consortium, August 2002.

[2] Karim Muller, "Semi-Automatic Construction of a Question Treebank," In Proceedings of the 4th International Conference on Language Resources and Evaluation, Lisbon, Portugal, 2004.

[3] H. Prüfer, "Neuer Beweis eines Satzes über Permutationen," Archiv fur Mathematik und Physik, 27: 142-144, 1998.

[4] Praveen R. Rao and Bongki Moon, "PRIX: Indexing and Querying XML Using Prüfer Sequences," In Proceedings of the 20th IEEE International Conference on Data Engineering, pp. 288-299, Boston, MA, March 2004.

[5] Mehmet Altinel and Michael J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," In Proceeding of the 26th VLDB Conference, pp. 53-64, Cairo, Egypt, September 2000.

[6] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang and Peter Fischer, "Path sharing and predicate evaluation for high-performance XML filtering," ACM Trans. Database Systems, Vol. 28, No. 4, pp. 467-516, 2003.

[7] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis and Rajeev Rastogi, "Efficient Filtering of XML Documents with XPath Expressions," In Proceedings of the 18th IEEE International Conference on Data Engineering, pp. 235-244, San Jose, CA, February 2002.

[8] Ashish Kumar Gupta and Dan Suciu, "Stream processing of XPath queries with predicates," In Proceeding of the 2003 ACM-SIGMOD conference, pp. 419-430, San Diego, CA, June 2003.

[9] T. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML Streams with Deterministic Automata and Stream Indexes," ACM Trans. on Database Systems, Vol. 29 No. 4, pp. 752-788, December 2004.

[10] Nicolas Bruno, Luis Gravano, Nick Koudas and Divesh Srivastava, "Navigation- vs. Index-Based XML Multi-Query Processing," In Proceedings of the 19th IEEE International Conference on Data Engineering, pp. 139-150, Bangalore, India, March 2003.

[11] Feng Tian, Berthold Reinwald, Hamid Pirahesh, Tobias Mayr and Jussi Myllymaki, "Implementing a Scalable XML Publish/Subscribe System Using a Relational Database System," In Proceeding of the 2004 ACM-SIGMOD Conference, pp. 479-490, Paris, France, June 2004.

[12] Quanzhong Li and Bongki Moon, "Indexing and Querying XML Data for Regular Path Expressions," In Proceeding of the 27th VLDB Conference, pp. 361-370, Rome, Italy, September 2001.
[13] N. Bruno, N. Koudas, D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," In Proceeding of the 2002 ACM-SIGMOD conference, pp. 310-321, Madison, WI, June 2002.
[14] David Megginson, Simple API for XML, http://sax.sourceforge.net/
[15] Apache Xerces C++ Parser. http://xml.apache.org/xerces-c/
[16] Angel Luis Diaz and Douglas Lovell, XML Generator. http://www.alphaworks. ibm.com/tech/xml-generator.

권 준 호

1999년 서울대학교 컴퓨터공학과 졸업 2001년 서울대학교 컴퓨터공학과 석사학위 취득. 2001년~현재 서울대학교 전기컴퓨터공학부 박사과정. 관심분야는 XML 인덱싱, XML 문서 필터링, 시공간 데이타베이스, 데이타마이닝

Praveen Rao

1999년 인도 University of Pune 컴퓨터공학과 졸업. 2001년 미국 University of Arizona 전산학과 석사학위 취득. 2002년~현재 University of Arizona 전산학과 박사과정. 관심분야는 XML indexing and query processing, XML based information dissemination, data stream processing과 large scale data processing

문 봉 기

정보과학회논문지 : 데이타베이스
제 33 권 제 3 호 참조

이 석 호

1964년 연세대학교 정치외교학과 졸업 1975년, 1979년 미국 텍사스대학교 전산학 석사와 박사학위 취득. 1979년~1982년 한국과학원 전산학과 조교수. 1982년~1986년 한국정보과학회 논문 편집위원장. 1986년~1989년 미국 IBM T.J. Watson 연구소 객원교수. 1988년~1990년 데이타베이스연구회 운영위원장. 1989~1991년 서울대학교 중앙교육연구전산원 원장. 1994년 한국정보과학회 회장. 1997년~1999년 한국학술진흥재단부설 첨단학술정보센터 소장. 1982년~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 데이타베이스, 멀티미디어 데이타베이스