대응효율성을 통한 변화 탐지 알고리즘의 성능 개선

이 석 균[†] · 김 동 아[††]

요 약

최근 웹 문서의 변조의 탐지, 버전 관리 등을 위한 XML/HTML 문서들에 대한 효과적인 실시간 변화탐지 알고리즘의 필요성이 증대하고 있다. 특히 대용량의 XML/HTML 문서들에 대한 실시간 변화탐지 응용들은 최소비용의 편집스크립트를 계산하는 알고리즘 보다는 실시간 처리가 가능한 빠른 휴리스틱 알고리즘들을 필요로 한다. 기존의 휴리스틱 알고리즘들은 실행속도는 빠르나 생성되는 편집스크립트의 질이 만족스럽지 못하다. 본 논문에서는 기존의 알고리즘 XyDiff와 X-tree Diff를 소개하고 이들 알고리즘들의 문제점들을 분석하고 문제점들을 개선한 알고리즘 X-tree Diff+를 제안한다. X-tree Diff+는 실행시간 측면에서 기존 알고리즘들과 유사하나 대응효율성에 기반한 대응과정의 개선을 통해 두 문서 간의 노드들의 대응률을 향상시킨 알고리즘이다.

키워드 : 계층구조 문서, XML, diff 알고리즘, 변화탐지

Improving Performance of Change Detection Algorithms through the Efficiency of Matching

Suk Kyoon Lee[†] · Dong-Ah Kim[††]

ABSTRACT

Recently, the needs for effective real-time change detection algorithms for XML/HTML documents are increased in such fields as the detection of defacement attacks to web documents, the version management, and so on. Especially, those applications of real-time change detection for large number of XML/HTML documents require fast heuristic algorithms to be used in real-time environment, instead of algorithms which compute minimal cost-edit scripts. Existing heuristic algorithms are fast in execution time, but do not provide satisfactory edit script. In this paper, we present existing algorithms XyDiff and X-tree Diff, analyze their problems and propose algorithm X-tree Diff+ which improve problems in existing ones. X-tree Diff+ has similar performance in execution time with existing algorithms, but it improves matching ratio between nodes from two documents by refining matching process based on the notion of efficiency of matching.

Key Words : Hierarchically-structured documents, XML, diff algorithm, Change Detection

1. 서 론

WWW의 확산으로 인해 정보의 접근이 편리해졌으나 정보의 양이 과도할 뿐 아니라 적절히 통제되고 있지 않아 WWW를 통해 필요한 정보 또는 유용한 정보를 찾는 것이 쉽지 않다. 특히 시간의 흐름에 따라 변화하는 정보들은 다양한 버전이 존재하게 되며, 특히 대용량의 정보들은 효율적인 관리가 중요한 문제로 부각되고 있다[9]. 또한 웹 페이지의 사용이 보편화됨에 따라 웹 페이지의 변조를 시도하는 해킹이 빈번해지고 있는데, 이러한 웹 페이지에 대한 변조 해킹은 빠른 시간 내에 탐지하는 게 매우 중요하다. 버전 관리와 웹 페이지의 변조 탐지의 문제는 두 문서의 변화 또는 차이를 탐지가 그 핵심이다[1, 18]. 따라서 본 논문에서는 기존의 실시간 변화 탐지 알고리즘들, XyDiff[9, 18]와 X-tree Diff[1, 6]를 소개하고 이들에 대한 비교 분석, 그리고 개선 방안을 제안하고 성능 분석의 결과를 보이고자 한다.

원본문서와 편집연산들이 적용된 대상문서에 대해 이들 문서의 차이는 일련의 편집연산들(즉 편집스크립트)로 표현되며 이 차이를 나타낼 수 있는 편집스크립트들은 무한하다. 두 문서 간의 차이를 나타내는 일련의 편집연산들(편집스크립트)의 생성 알고리즘에 대한 연구들은 1970년대부터 지속적으로 이루어졌다[1, 2, 5, 6, 7, 9, 10, 11, 15, 17]. 초기 연구들로는 Wagner와 Fisher의 알고리즘[14] 등이 있는데 이들은 주로 두 텍스트 파일에 대한 최소 비용의 편집스크립트를 구하고 있다. 대표적인 텍스트 기반의 변화 탐지 프로그램 GNU의 diff는 LCS(Longest Common Subsequence) 알고리즘[8]에 기반하고 있으며 이는 또한 프로그램 버전 관리 프로그램인 CVS[4]에 사용되고 있다.

※이 연구는 2006년도 단국대학교 대학연구비의 지원으로 연구되었음.
† 종신회원 : 단국대학교 정보컴퓨터학부 교수
†† 준 회 원 : 단국대학교 컴퓨터과학전공 겸임교수
논문접수 : 2006년 1월 31일, 심사완료 : 2006년 3월 20일

트리구조 문서에 대한 변화 탐지 연구들[2, 10, 15, 16, 17, 18, 19]도 주로 최소 비용의 편집 스크립트의 생성에 관심이 집중되었다. 일반적인 트리 구조의 데이터에 대한 변화 탐지 알고리즘은 NP-Hard 문제[15]로 트리 구조 문서에 대한 최소 비용의 변화 탐지 알고리즘들은 텍스트 기반의 변화 탐지에 비해 훨씬 높은 비용이 발생한다. Selkow 알고리즘[17], Tai 알고리즘[10], Zhang과 Shasha 알고리즘[11], Chawathe의 MH-Diff[15] 등 여러 가지 알고리즘들이 존재하나 높은 비용으로 인해 실시간의 대용량 처리에는 적합하지 않다.

최근의 트리구조 문서에 대한 변화 탐지 연구는 현실적인 필요에 의한 것이다. WWW의 XML/HTML 문서들의 변화(차이)에 대한 편집스크립트를 생성하는 것으로 최소 비용의 편집 스크립트를 위한 알고리즘 대신, 실시간의 빠른 처리를 위한 휴리스틱 알고리즘의 개발에 관심이 집중되고 있다[1, 5, 6, 9, 18, 19]. XyDiff 알고리즘[9, 18]은 대용량 XML 문서의 빠른 변화 탐지를 목적으로 개발되어 XML 문서 데이터로부터 능동형 웨어하우스 구축을 위한 Xyleme 프로젝트[19]에 사용되었다. XyDiff는 대응과정에 해시를 사용하는 알고리즘으로 트리의 노드 수를 n이라 할 때, 실행시간이 $O(n*log(n))$의 성능을 보이는 휴리스틱 알고리즘이다. 한편, X-tree Diff는 대용량의 웹 페이지들을 모니터링하면서 해킹을 통한 웹 페이지의 변조를 탐지하기 위한 시스템 WIDS(Web Document Intrusion Detection System)에 사용하기 위해 개발된 알고리즘이다[1, 6]. X-tree Diff 알고리즘도 비교의 효율성을 위해 해시를 사용하나, XyDiff에 비해 알고리즘이 간결하고 실행시간이 $O(n)$의 성능을 보인다.

XyDiff와 X-tree Diff 등의 휴리스틱 알고리즘은 실행 속도는 빠르나, 최소 비용을 계산하는 알고리즘들에 비해 대응의 질이 다소 떨어지는 특성이 있다. 본 논문에서는 이들 휴리스틱 알고리즘들을 분석하고 이들을 개선한 X-tree Diff+ 알고리즘을 제안한다. X-tree Diff+는 실행시간의 성능은 $O(n)$을 유지하면서 XyDiff와 X-tree Diff에 비해 대응의질을 향상시킨다.

XyDiff의 대응의 질을 개선한 알고리즘으로 X-Diff[20]가 있다. X-Diff도 해싱과 다이내믹 프로그래밍을 통해 최소비용의 편집거리(edit distance)를 계산한다. 대응의 질은 좋아졌으나 이로 인한 실행시간이 증가한다. deg(T)를 트리의 최대 자식노드 수라하고 T를 트리의 전체 노드수라 할 때, 비용은 $O(T_1 \times T_2 \times \max\{\deg(T_1),\deg(T_2)\} \times \log_2(\max\{\deg(T_1),\deg(T_2)\}))$이다. 따라서 본 논문에서는 비용의 급속한 증가로 인해 X-Diff에 대한 비교 분석은 제외한다.

논문의 구성은 다음과 같다. 2절에서는 X-tree Diff 알고리즘과 XyDiff 알고리즘을 소개하고 문제점들을 제기한다. 3절에서는 이들 문제점들을 개선한 X-tree Diff+를 제안하고, 4절에서는 X-tree Diff+의 비용이 $O(n)$임을 보이고 기존 알고리즘들과의 성능 비교 자료를 제시한다. 끝으로 5절에서 결론 및 미래 연구를 소개한다.

2. 기존의 휴리스틱 트리 Diff 알고리즘

본 절에서는 문제점 분석 목적을 위해 예제 중심으로 설명한다.

2.1 X-tree Diff 알고리즘의 개요

X-tree Diff는 변화 탐지의 대상이 되는 두 XML 문서를 각각 X-tree T와 T'로 변환한 후, 3 단계로 구성된 대응 과정을 거친 후 변화에 대한 편집 스크립트를 생성한다. 우선 X-tree의 노드 구조와 지원 연산을 설명한다.

2.1.1 X-tree 구조 및 지원 연산

X-tree 노드의 구조는 (그림 1)와 같다[1, 6]. 단 트리 구조를 위한 필드들은 편의 상 생략한다.

Type필드에는 엘리먼트 노드는 1 그리고 텍스트 노드는 0의 값이 저장되며, Label필드에는 엘리먼트 노드의 경우 엘리먼트 노드의 레이블 이름이, 텍스트 노드의 경우는 '#TEXT'의 값이, 그리고 Value 필드에는 엘리먼트 노드의 경우 속성과 속성 값 쌍의 리스트가 문자열로 저장된다. Index 필드는 같은 부모노드 내에 동일한 Label을 갖는 자식노드들 간의 순서로 같은 Label을 갖는 형제노드들을 식별을 위해 사용된다. (그림 2)(a)의 XML 문서에 대한 Label, Type, Value, Index 필드들의 사용 예가 (그림 2)(b)의 X-tree에서 제시되고 있다.

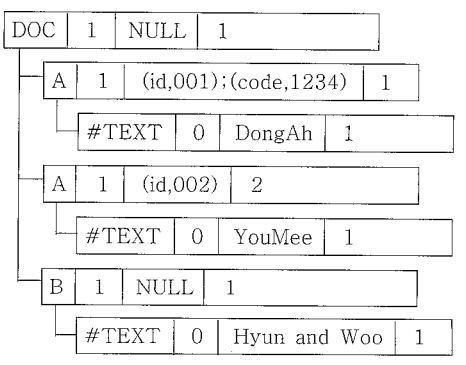
한편, 노드와 트리간의 효율적인 비교를 위해 해시가 사용되는데, 이를 위해 nMD와 tMD 필드가 사용된다. 이때 해시

Label	Type	Value	Index	nMD	tMD	nPtr	Op

(그림 1) X-tree 노드의 구조

```
<?xml version="1.0" encoding="iso-8859-1"?>
<DOC>
  <A id="001" code="1234">
    DongAh
  </A>
  <A id="002">
    YouMee
  </A>
  <B>
    Hyun and Woo
  </B>
</DOC>
```

(a)



(b)

(그림 2) 간단한 XML 문서와 X-tree 모델의 표현

함수로는 MD4 해시 알고리즘[13]이 사용되며 설명의 편의상 필드의 이름이 X-tree 노드에 대해 필드 값을 반환하는 함수로 사용된다. 노드 τ_i에 대한 nMD와 tMD 필드 값은 $nMD(\tau_i)$와 $tMD(\tau_i)$로 표현되며 이들 함수는 다음과 같이 정의된다. 단 MD는 MD4 해시함수, $\cup$는 문자열 결합연산자, $C_x(\tau_i)$는 τ_i의 x번째 자식노드를 반환하는 함수임.

$$nMD(\tau_i) = string(MD(\ Label(\tau_i) \cup\ Value(\tau_i)))$$
$$tMD(\tau_i) = string(\ MD(nMD(\tau_i) \bigcup_{x=1}^{n} \mathrm{tMD}(\ C_x(\tau_i))))$$

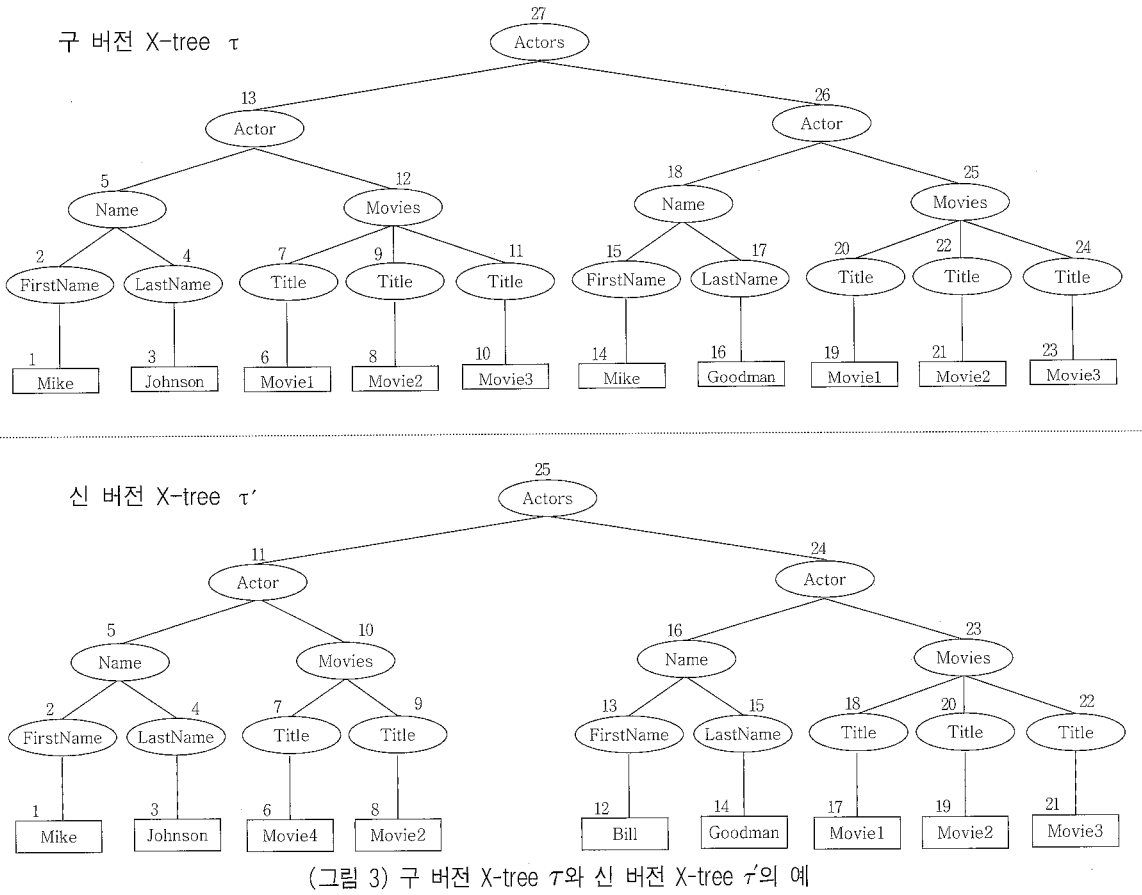
X-tree Diff에서 *대응*의 개념은 Selkow의 노드의 대응[17]의 개념을 확장한 것으로, X-tree T의 노드 τ_i와 T'의 노드 τ'_j의 대응은 $\tau_i \rightarrow \tau'_j$로 표현한다. $ST(\tau_i)$를 노드 τ_i를 루트로 하는 서브트리를 반환하는 함수라 할 때, τ_i를 루트로 하는 서브트리와 τ'_j를 루트로 하는 서브트리의 대응은 $ST(\tau_i) \rightarrow ST(\tau'_j)$로 표현한다. $tMD(\tau_i) = tMD(\tau'_j)$인 경우 두 서브트리는 대응, 즉 $ST(\tau_i) \rightarrow ST(\tau'_j)$로 표현된다. 이 때 서브트리 $ST(\tau_i)$에 속한 각 노드는 서브트리 $ST(\tau'_j)$의 상응되는 노드와 일대일로 대응하게 된다.

X-tree Diff에서는 삽입, 삭제, 갱신연산은 물론 대부분의 기존 연구들과는 달리 이동연산을 지원한다. 원본 X-tree T에 대해 삽입, 삭제, 갱신, 이동연산은 <표 1>과 같이 정의된다.

편의상 원본 트리의 노드 τ_i가 대상 트리에 아무런 변화 없이 남아있는 경우를 위해 $N(\tau_i)$, 즉 공연산(Null operation)의 개념을 사용한다. X-tree 노드의 nPtr 필드에는 대응 노드의 주소가, Op 필드에는 대응에 적용되는 편집연산이 저장된다. 즉, X-tree T의 임의의 노드 τ_i가 X-tree T'의 노드 τ'_j와 대응되면, 각각의 nPtr 필드에는 대응 노드의 주소가, Op 필드에는 갱신, 이동 그리고 공연산 중 하나의 편집연산이 저장된다. 특별한 언급 없이 대응된다고 하면 공연산

〈표 1〉 X-tree Diff에서의 지원 편집 연산과 그 의미

편집연산	연산 의미
삽입연산 $Ins(l, v, \tau_i, k)$	Label 값 l과 Value 값 v를 갖는 노드를 생성한 후, 이를 노드 τ_i의 k번째 자식노드로 삽입한다.
삭제연산 $Del(\tau_i)$	노드 τ_i를 루트로 하는 서브트리를 트리 T로부터 삭제한다.
갱신연산 $Upd(\tau_i, v')$	노드 τ_i의 Value 필드의 값을 v'으로 갱신한다.
이동연산 $Mov(\tau_i, \tau_j, k)$	노드 τ_i을 루트노드로 하는 서브트리 $ST(\tau_i)$를 노드 τ_j의 k번째 자식노드로 이동시킨다.



(그림 3) 구 버전 X-tree T와 신 버전 X-tree T'의 예

이 저장됨을 의미한다. 또한 본 논문에서는 대응 과정의 설명을 위해 $M(t)$는 노드의 t의 대응 노드를 반환하며, $C_x(t)$는 노드 t의 x번째 자식노드를, $P(t)$는 노드 t의 부모노드를 반환하는 함수로 정의한다.

2.1.2 X-tree Diff 알고리즘의 소개

본 절에서는 X-tree Diff 알고리즘을 3단계로 나누어 설명한다. 이때 트리의 생성과 자료구조의 초기화 과정은 생략한다. (그림 3)은 원본 X-tree T와 대상 X-tree T'의 하나의 예로 엘리먼트 노드는 타원으로 텍스트 노드는 직사각형으로 표시되고 각 노드에는 트리의 후위 순회수(postfix)가 식별자로 주어져있다. 즉, T의 i번째 후위 순회의 노드는 τ_i로 표기하고, 설명의 편의 상 Value 필드들은 고려하지 않는다.

1 단계 - 유일한 동일 서브트리(identical subtree) 대응

X-tree Diff 알고리즘은 우선 X-tree로 표현된 트리 T와 트리 T' 사이의 존재하는 일대일(1:1) 관계의 동일 서브트리(identical subtree)를 찾아 대응시킨다. 이 때 다대일, 일대다 또는 다대다 관계의 동일 서브트리의 쌍은 현 단계에서 대응시키지 않는다. 서브트리의 비교에는 서브트리의 루트 노드의 tMD 필드가 사용되므로 효율적인 비교가 가능하다.

(그림 3)의 예에서 T의 서브트리 $ST(\tau_5)$, $ST(\tau_{17})$은 T'에 각각 유일한 동일 서브트리 $ST(\tau'_5)$와 $ST(\tau'_{15})$가 존재하므로 $ST(\tau_5)\rightarrow ST(\tau'_5)$와 $ST(\tau_{17})\rightarrow ST(\tau'_{15})$의 서브트리 대응이 이루어진다. 서브트리 $ST(\tau_{12})$와 $ST(\tau_{25})$는 동일한 서브트리들로 둘 다 서브트리 $ST(\tau'_{23})$와 대응 가능하나, 이러한 다대일, 일대다 또는 다대다 관계의 동일 서브트리의 쌍들에 대한 대응 결정은 나중으로 미루게 되는데 이를 *모호한 대응의 연기 규칙*이라 한다.

2 단계 - 대응의 상향 확대(bottom-up propagation)

2 단계는 1 단계에서 대응이 결정된 서브트리들로부터 조상노드로 대응을 확대해가는 과정이다. 현재까지 대응된 서브트리 쌍의 각 루트의 부모노드의 Label들을 비교해서, 같은 경우 대응을 부모노드로 확대한다. (그림 3)에서 1 단계에서 결정된 대응 $\tau_5\rightarrow\tau'_5$은 그 부모노드들의 Label이 같으므로 부모노드들 간의 대응 $\tau_{13}\rightarrow\tau'_{11}$으로 확대되며, 다시 이들의 부모노드들의 Label 또한 같으므로 대응은 $\tau_{27}\rightarrow\tau'_{25}$으로 확대된다. 마찬가지로 대응 $\tau_{17}\rightarrow\tau'_{15}$은 $\tau_{18}\rightarrow\tau'_{16}$, 또한 $\tau_{26}\rightarrow\tau'_{24}$으로 상향 확대된다.

3 단계 - 대응의 하향 확대(top-down propagation)

3 단계에서는 X-tree T를 깊이우선탐색(depth-first traversal) 순서로 접근하면서 대응이 결정된 노드의 자식 노드들 중 아직 대응이 결정되지 않은 노드들을 대응시킨다. 우선 대응된 노드의 쌍 τ_i와 τ'_j의 (대응이 결정되지 않은) 자식노드들에 대해 tMD값이 같은 자식노드들의 쌍(τ_{im}, τ'_{jn})이 있으면 이들 노드들을 루트로 하는 서브트리 쌍($ST(\tau_{im})$, $ST(\tau'_{jn})$)을 대응시킨다. tMD 값이 같은 노드 쌍이 더 이상 없고, Label이 같은 노드의 쌍이 존재할 때(즉, $\text{Label}(\tau_{im}) = \text{Label}(\tau'_{jn})$의 경우)는 먼저 그 노드 쌍의 Value 필드의 값을 확인한다. Value 필드의 값도 같을 경우($\text{Value}(\tau_{im}) = \text{Value}(\tau'_{jn})$의 경우)에는 이들 노드 쌍($\tau_{im}$, τ'_{jn})을 공연산으로 대응시키고 Value 필드 값이 다를 경우는 이들 노드 쌍을 갱신연산으로 대응시킨다. 이 때 동일 tMD 값으로 대응되는 경우는 1 단계에서 일대다, 다대일 또는 다대다의 대응이 가능하여 모호한 대응 연기 규칙에 의해 그 대응이 연기된 경우이다.

(그림 3)의 X-tree T를 깊이우선탐색 순으로 접근 시, $\tau_{13}\rightarrow\tau'_{11}$으로 대응된 노드의 쌍($\tau_{13}$, τ'_{11})에서 대응이 결정되지 않은 자식노드의 쌍 (τ_{12}, τ'_{10})은 같은 Label을 가지고 있어 (Value 필드 값은 없음에 주의) $\tau_{12}\rightarrow\tau'_{10}$의 공연산 대응이 결정된다. 이와 같이 대응된 노드 $\tau_{12}\rightarrow\tau'_{10}$의 자식노드들을 비교하면 유사한 이유로 $\tau_7\rightarrow\tau'_7$의 공연산 대응(동일 Label), $\tau_6\rightarrow\tau'_6$의 갱신연산 대응(텍스트 노드는 Label이 '#TEXT'임), 그리고 $ST(\tau_9)\rightarrow ST(\tau'_9)$의 대응(동일한 tMD)이 결정된다. 마찬가지로 $\tau_{18}\rightarrow\tau'_{16}$로부터 $\tau_{15}\rightarrow\tau'_{13}$의 공연산 대응 그리고 $\tau_{14}\rightarrow\tau'_{12}$의 갱신연산 대응이 결정되며, $\tau_{26}\rightarrow\tau'_{24}$로부터 $ST(\tau_{25})\rightarrow ST(\tau'_{23})$의 공연산 대응이 결정된다. 이 때 $ST(\tau_{25})\rightarrow ST(\tau'_{23})$의 대응은 $ST(\tau_{12})$와 $ST(\tau_{25})$이 동일 tMD 값을 가지고 있어 1 단계에서 대응이 연기되었던 것이다.

이후 X-tree Diff에서는 앞에서 설명한 단계들에서 생성된 대응들로부터 편집스크립트를 생성한다. 간단히 설명하면, X-tree T에서 대응이 이루어지지 않은 노드들에 대해서는 삭제연산을, 반대로 T'에서 대응되지 않고 남아있는 노드들에 대해서는 삽입연산을 생성한다. 대응된 노드에 대해서, Op 필드의 값에 따라 편집연산이 생성된다. (그림 3)의 예에서는 $\{ Del(\tau_{10}),\ Del(\tau_{11}),\ Upd(\tau_6,\ \text{'Movie1'},\ \text{'Movie4'}),\ Upd(\tau_{14},\ \text{'Mike'},\ \text{'Bill'}) \}$의 편집스크립트가 생성된다.

2.2 XyDiff 알고리즘의 개요

본 절에서는 Xyleme 프로젝트[19]에서 사용된 XyDiff 알고리즘[9, 18]을 예제를 통해 설명한다.

2.2.1 XyDiff의 트리 구조

XyDiff 알고리즘은 X-tree Diff 알고리즘과 같이 서브트리의 정보에 대한 해시값을 대응 과정에 사용한다. 사용되는 트리의 노드 구조는 DOM을 확장한 형태로 (그림 4)에 제시되어 있다. 노드의 XID 필드에는 노드의 후위 순회수(postfix)가 저장되어 노드의 식별에 사용된다. ID속성이 있는 엘리먼트의 경우, 노드의 hasIDAttr 필드가 TRUE로 설정되며 Weight 필드에는 노드의 가중치가 저장되어 대응 과정에 사용된다. 노드의 가중치는 다음과 같이 정의된다.

XID	hasIDAttr	Weight	OwnHash	SubtreeHash	MatchID	Event

(그림 4) XyDiff 노드의 구조

- 텍스트 노드의 가중치 = $1+log$(문자열의 길이)
- 엘리먼트 노드의 가중치 = $1+sum$(자식노드의 가중치)

OwnHash 필드는 X-tree Diff의 nMD와 유사한 개념으로, 엘리먼트 노드의 레이블과 노드의 ID속성 값을 키로 얻은 해시값을 저장한다. ID속성이 없는 엘리먼트의 경우 노드의 레이블으로 해시값을 계산한다. 한편, 텍스트 노드의 OwnHash 필드에는 0이 사용된다. SubtreeHash 필드는 X-treeDiff의 tMD와 유사한 개념으로 엘리먼트 노드의 경우 자신의 OwnHash 필드값과 자식노드들의 SubtreeHash 필드값들을 키로 계산된 해시값이, 단말노드인 텍스트 노드의 경우 텍스트 내용을 키로 계산된 해시값이 저장된다. XyDiff에서는 대응 노드의 정보를 유지하기 위해 대응 노드의 XID를 MatchID 필드에 저장하며 이 때 적용되는 편집 연산은 Event 필드에 저장된다.

2.2.2 XyDiff 알고리즘의 소개

XyDiff는 XML의 ID 속성을 사용하여 노드의 대응을 찾는 것으로 시작하여 전체 3 단계로 구성된다. 다음의 각 단계별 처리의 내용은 XyDiff 알고리즘의 구현내용[18]을 중심으로 정리하였다. 앞에서와 마찬가지로 XyDiff 트리의 초기화 및 기타 자료 구조 생성을 위한 단계는 생략한다.

1 단계 - ID 속성을 이용한 대응과 상향 확대

트리 τ'를 깊이우선으로 순회하면서 ID속성을 갖는 노드 τ'_j를 만나면, 트리 τ에서 같은 OwnHash 값의 노드 τ_i를 찾아 τ'_j와 대응시킨다. ID 속성을 갖는 노드들에 대한 대응을 모두 결정한 후, 트리 τ'을 다시 후위 순회하면서 결정된 대응들을 조상노드로 확대시켜 나간다. 조상노드로의 대응의 확대 과정은 X-tree Diff와 유사하다. 대응된 노드의 쌍으로부터 부모노드의 OwnHash 값을 비교하고 두 부모노드의 OwnHash가 서로 같을 경우, 부모노드를 대응시킨다.

그러나 다수의 자식노드들이 있는 부모노드의 경우, 자식노드의 대응을 부모노드로의 대응을 확대하는 과정에서 모호성이 발생할 수 있다. 즉 τ'_j의 p, q번째 자식 노드를 각각 $C_p(\tau'_j)$, $C_q(\tau'_j)$라 할 때 $P(M(C_p(\tau'_j)))$와 $P(M(C_q(\tau'_j)))$, 즉 τ'_j의 임의 두 자식 노드들의 대응 노드들, 그리고 그 대응 노드들의 부모노드들이 같지 않을 경우, τ'_j를 트리 τ의 노드 $P(M(C_p(\tau'_j)))$와 $P(M(C_q(\tau'_j)))$ 중 어느 노드와 대응을 시켜야할지 불분명하다.

(그림 5)의 예에서 파선으로 표현한 대응 $\tau_{14}\rightarrow\tau'_{14}$, $\tau_{21}\rightarrow\tau'_{15}$, $\tau_{22}\rightarrow\tau'_{16}$, $\tau_{23}\rightarrow\tau'_{17}$ 그리고 $\tau_{24}\rightarrow\tau'_{24}$은 ID 속성에 의해 대응된 노드 쌍을 나타낸다. 한편, 노드 τ_{15}, τ_{25}, τ'_{18} 그리고 τ'_{25}는 모두 같은 OwnHash 값을 갖는다고 가정하면, τ'_{18}는 $\tau_{14}\rightarrow\tau'_{14}$을 통해 τ_{15}와 대응이 가능하며 또한 $\tau_{21}\rightarrow\tau'_{15}$를 통해 τ_{25}와도 대응이 가능하다.

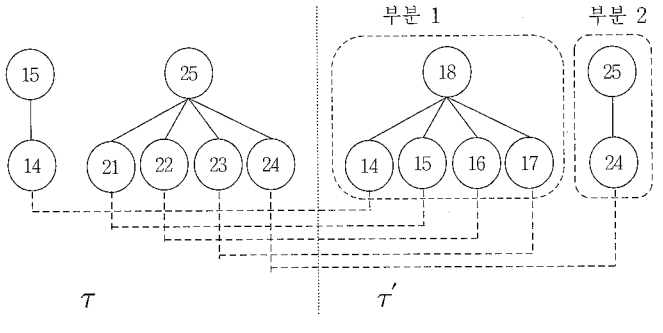
XyDiff는 이러한 모호성을 해결하기 위해 노드의 가중치를 사용한다. 즉 τ'_j와 대응 가능한 임의 노드 τ_i에 대해, $\tau_i = P(M(C_k(\tau'_j)))$를 만족하는 τ'_j의 모든 자식노드들 $C_k(\tau'_j)$의 가중치의 합을 계산하여 이들 값이 가장 큰 τ_i를 τ'_j의 대응 노드로 결정한다.

(그림 5)(b)는 모든 단말노드의 가중치가 1이라 할 때 계산된 이들 가중치의 합을 보인다. XyDiff의 상향 확대 과정은 트리 τ'를 후위 순회하며 모든 자식노드의 대응을 확인한 후 부모노드의 대응을 결정한다. 따라서 (그림 5)(a)의 '부분 1'에서 가중치의 계산으로 $\tau_{25}\rightarrow\tau'_{18}$의 대응이 결정되나 '부분 2'에서 τ'_{25}는 대응 가능한 τ_{25}가 이미 대응이 결정되어 대응되지 않은 상태로 남게 된다.

2 단계 - 루트노드로부터 대응의 하향 확대

본 단계에서는 루트로부터 대응의 확대 과정에서 트리 τ와 트리 τ' 사이에 존재하는 동일 서브트리 쌍을 대응시킨다. 트리 τ와 τ'사이에는 다수의 동일 서브트리들이 존재한다면 어떤 대응을 결정할 지 불분명해지는데, 이러한 경우 XyDiff에서는 그 각각의 부모노드들, 2대 조상노드들, 3대 조상노드들의 대응 여부를 고려해서 대응을 결정한다.

X-tree Diff의 3 단계에서는 대응된 부모노드의 자식노드



임시 테이블(부분 1)

XID	가중치
15	1.0
25	3.0

임시 테이블(부분 2)

XID	가중치
25	1.0

(b)

(그림 5) 대응노드의 상향 확대(propagation)

들 사이에 동일 서브트리가 있을 때 서로 대응시키므로 동일 서브트리의 대응에 부모노드의 대응 여부가 영향을 준다. 그러나 XyDiff에서는 부모노드들이 대응되었을 때는 물론, 부모노드들이 대응되지 않은 경우에는 2대 조상노드들 사이에 대응관계가 있을 경우, 2대 조상노드들도 대응이 되지 않은 경우는 3대 조상노드들이 대응되는 경우까지도 동일 서브트리의 대응에 고려한다. 보다 정확히 설명하면 동일 서브트리의 대응에 고려할 조상노드의 대수를 무한정 확대하는 것이 아니라 대응 여부를 확인할 조상노드의 대수를 $3+\frac{w}{w_0}\cdot log(w_0)$로 한정한다. 이 때 w는 방문 노드 τ'_j의 가중치이며 w_0는 트리 τ'의 루트노드의 가중치를 의미한다. 한편, 이렇게 결정된 서브트리의 루트노드로부터 대응된 조상노드까지 경로 상에 있는 대응이 결정되지 않은 노드들은 OwnHash 값의 비교를 통해 차례로 대응시켜 나간다. 그러나 레이블이 다른 조상노드를 만나면 순서대로 대응시켜 나가던 것을 중단한다.

(그림 3)의 예를 통해 본 과정을 설명한다. XyDiff에서는 루트노드는 변하지 않는다고 가정하기 때문에 대응 $\tau_{27}\rightarrow\tau'_{25}$이 형성된다. 트리 τ'을 너비우선으로 순회하면서 노드 τ'_{11} 방문시 τ에는 $ST(\tau'_{11})$와 동일한 서브트리가 존재하지 않아 다음 노드 τ'_{24}를 방문한다. τ'_{24}도 같은 이유로 다음 노드 τ'_5를 방문한다. τ'_5는 동일 서브트리가 존재(즉, $ST(\tau'_5)=ST(\tau_5)$)하므로 부모노드의 대응을 확인하나 τ'_{11}가 대응이 결정되지 않은 노드로 다시 2대 조상 τ'_{25}의 대응 여부를 확인한다. τ'_{25}는 루트노드로 루트노드 τ_{27}와 이미 대응되어 있어, 즉 대응 $P(P(\tau_5))\rightarrow P(P(\tau'_5))$이 이미 존재하고 있어 동일 서브트리 대응을 결정한다. 그리고 경로 상에 있는 노드들의 대응 즉 $\tau_{13}\rightarrow\tau'_{11}$을 결정한다. 본 단계를 통해 $ST(\tau_{12})\rightarrow ST(\tau'_{23})$, $ST(\tau_{22})\rightarrow ST(\tau'_9)$, $\tau_{25}\rightarrow\tau'_{10}$, $ST(\tau_{17})\rightarrow ST(\tau'_{15})$, $\tau_{18}\rightarrow\tau'_{16}$, $\tau_{26}\rightarrow\tau'_{24}$ 등의 서브트리 대응과 노드 대응이 이루어진다.

3 단계 - 국소적 최적화(Peephole Optimization)

XyDiff에서 대응을 생성하는 마지막 단계로, 트리 τ를 깊이우선으로 순회하면서 다음의 과정을 반복 수행한다. 방문한 노드 τ_i가 대응이 결정된 경우, τ_i의 자식노드들 중 대응되지 않은 노드들과 τ_i의 대응 노드 $M(\tau_i)$의 자식노드들 중 대응되지 않은 자식노드들 중 동일한 그러면서도 일대일 대응관계에 있는 레이블을 갖는 노드의 쌍을 찾아 대응시킨다.

(그림 2-3)의 예에서 $\tau_{18}\rightarrow\tau'_{16}$로 대응된 노드 τ_{18}과 τ'_{16}은 각각 비-대응 자식노드 τ_{15}와 τ'_{13}을 갖는다. τ_{15}와 τ'_{13}은 일대일 관계의 동일한 레이블을 갖는 노드이므로 이 단계에서 대응을 시킨다. 한편, $\tau_{25}\rightarrow\tau'_{10}$로 대응된 τ_{25}과 τ'_{10}의 경우에는 τ_{25}의 비-대응 자식노드들 τ_{20}과 τ_{24}은 τ'_{10}의 비-대응 자식노드 τ'_7과 대응시킬 수 없다.

이후 XyDiff가 앞에서 설명한 단계들에서 결정된 대응들로부터 편집스크립트를 생성하는 것은 X-tree Diff에서와 유사하다. (그림 3)의 예로부터 편집스크립트 {$Del(\tau_{19})$, $Del(\tau_{20})$, $Del(\tau_{23})$, $Del(\tau_{24})$, $Mov(\tau_{12}, \tau_{26}, 2)$, $Mov(\tau_{25}, \tau_{13}, 2)$, $Ins('Title', NULL, \tau'_{10}, 1)$, $Ins('\#TEXT', 'Movie4', \tau'_7, 1)$}을 본 과정을 통해 생성한다.

XyDiff는 X-tree Diff와 같이 n을 신·구 버전 문서 크기의 합이라 하면, 전 과정을 통해 $O(n)$에 바운드된다.

2.3 기존 휴리스틱 알고리즘의 문제점

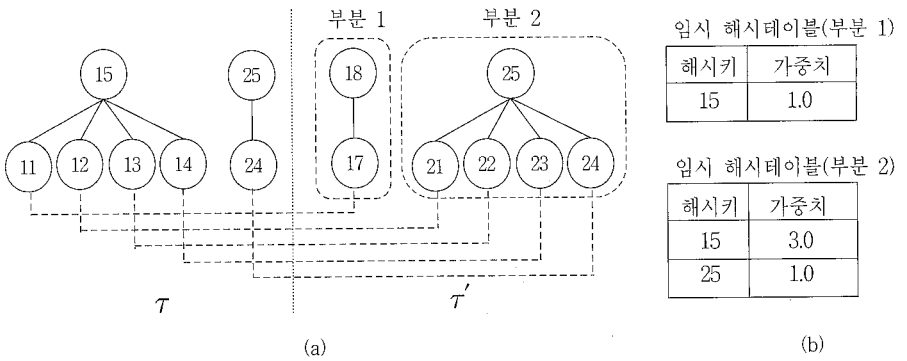
휴리스틱 알고리즘들은 그 성격 상 높은 비용이 발생하는 전역적인 분석을 통해 대응을 생성할 수는 없다. 그러나 대응 과정에서 적절한 국지적인 문맥의 분석, 즉 인접 노드들의 대응정보에 대한 체계적인 분석은 좀더 나은 대응을 생성할 수 있다. XyDiff는 대응의 확대과정에서 판단이 모호한 경우 가중치를 사용하고 X-tree Diff는 일대일이 아닌 동일 서브트리 쌍의 경우, 즉 대응이 모호한 경우 나중 단계로 대응을 연기시키는 정도이다. 본 절에서는 XyDiff에서 대응과정에서 가중치를 사용하는 방법의 문제점들을 보인다. 본 절의 내용은 가중치를 사용하지 않은 X-tree Diff에도 동일하게 적용될 수 있다.

문제점 1

XyDiff의 1단계에서는 대응의 확대 시 트리 τ'을 후위 순회하며 모든 자식노드의 대응을 확인한 후 부모노드의 대응을 결정한다. 이 전략으로부터 발생하는 문제를 보자. (그림 6)는 (그림 5)의 서브트리들의 순서만을 변경시킨 예로서 (그림 5)에서와 같이 파선은 같은 ID 속성 값을 갖는 노드들을 나타내어 ID속성에 의해 $\tau_{11}\rightarrow\tau'_{17}$, $\tau_{12}\rightarrow\tau'_{21}$, $\tau_{13}\rightarrow\tau'_{22}$, $\tau_{14}\rightarrow\tau'_{23}$ 그리고 $\tau_{24}\rightarrow\tau'_{24}$ 대응들이 결정되어있고 노드 τ_{15}, τ_{25}, τ'_{18} 그리고 τ'_{25}는 모두 같은 OwnHash 값을 가지고 있다고 가정하자.

이때 τ'_{18}은 대응 $\tau_{11}\rightarrow\tau'_{17}$을 통해 τ_{15}와 대응이 가능하며, τ'_{25}은 대응 $\tau_{12}\rightarrow\tau'_{21}$, $\tau_{13}\rightarrow\tau'_{22}$, $\tau_{14}\rightarrow\tau'_{23}$을 통해 τ_{15}와 그리고 $\tau_{24}\rightarrow\tau'_{24}$를 통해 τ_{25}와도 대응이 가능하다. 그러나 후위 순회의 방문 순서에 따라 τ'_{18}의 대응 결정은 τ'_{25}를 방문하기 전에 이루어져야 하므로 τ'_{18}에 대한 대응 $\tau_{15}\rightarrow\tau'_{18}$이 결정되고 τ'_{25}는 τ_{25}보다 τ_{15}와의 대응 가능 가중치가 높음에도 불구하고 $\tau_{25}\rightarrow\tau'_{25}$의 대응이 결정된다.

이들 대응으로부터 생성된 편집스크립트는 {$Mov(\tau_{12}, \tau_{25}, 1)$, $Mov(\tau_{13}, \tau_{25}, 2)$, $Mov(\tau_{14}, \tau_{25}, 3)$}으로 (그림 5)로부터의 편집스크립트 {$Mov(\tau_{14}, \tau_{25}, 1)$, $Mov(\tau_{24}, \tau_{15}, 1)$}보다 비효율적이다. 이는 후위 순회를 통해 이미 결정된 대응을 나중에 발견된 보다 나은 대응으로 변경시킬 수 없음에서 발생하는 문제이다.



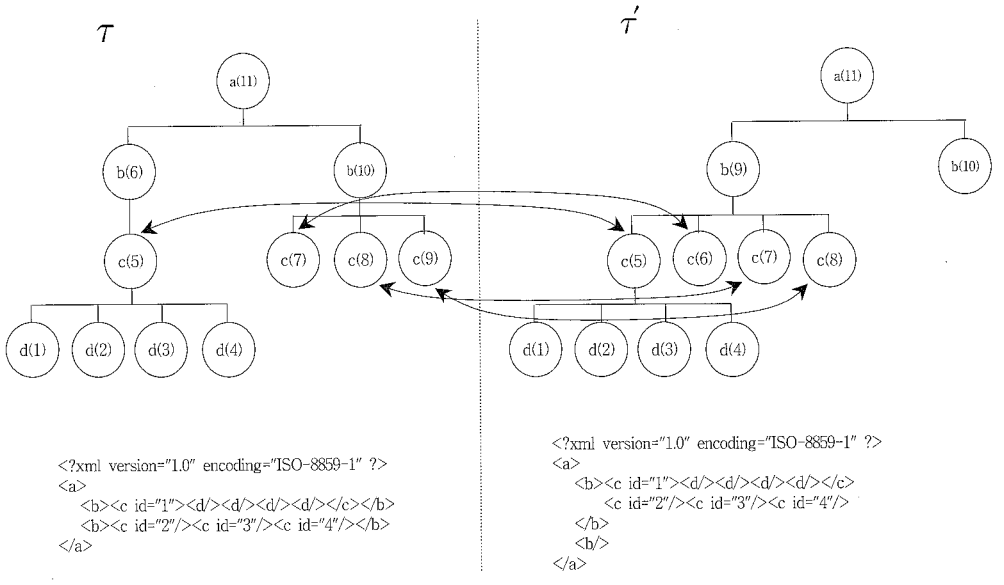
임시 해시테이블(부분 1)

해시키	가중치
15	1.0

임시 해시테이블(부분 2)

해시키	가중치
15	3.0
25	1.0

(그림 6) 대응의 상향 확대 과정의 문제(1)



(그림 7) 대응의 상향 확대 과정의 문제(2)

문제점 2

XyDiff는 대응의 상향 확대 과정에서 자식노드들의 가중치를 사용한다. (그림 7)에서는 자식노드의 가중치를 사용하여 대응을 확대할 때 발생하는 문제를 보인다. (그림 7)에서는 노드의 식별을 위해 노드의 레이블과 레이블 우측의 ( ) 안에는 노드의 후위 순회수를 표시하고, ID 속성에 의해 대응되는 노드의 쌍은 파선 화살표로 나타낸다.

ID 속성에 의해 대응된 노드들로부터 대응의 확대 과정을 고려하면 노드 τ'_9는 같은 레이블이 'b'인 τ_6과 τ_{10}이 대응 가능하다. 그림에서 볼 때 모든 노드들이 엘리먼트 노드들이어서, 노드 τ'_9가 τ_6과 대응될 가중치는 5.0, τ_{10}과 대응될 가중치는 3.0이 된다. 따라서 $\tau_6 \rightarrow \tau'_9$의 대응이 결정된다. 트리 T와 T'의 노드들이 이후 과정을 통해 대응이 결정된다고 할 때, 이들 대응으로부터 편집스크립트 {$Mov(\tau_7, \tau_6, 2)$, $Mov(\tau_8, \tau_6, 3)$, $Mov(\tau_9, \tau_6, 4)$}가 생성된다. 그러나 τ'_9가 τ_{10}과 대응되고, 트리 T와 T'에 속한 모든 노드가 이후 과정을 통해 대응된다고 가정할 경우, 편집스크립트 {$Mov(\tau_5, \tau_{10}, 1)$}는 위에서 생성된 편집스크립트보다 효율적이다.

3. X-tree Diff+ 알고리즘

X-tree Diff+는 X-tree Diff 알고리즘과 XyDiff을 기반으로 앞에서 언급한 문제점들을 개선한 알고리즘이다. X-treeDiff와 마찬가지로 $O(n)$의 비용을 유지하면서 복사연산을 추가하였다. X-tree Diff+ 알고리즘은 X-tree Diff와 단계들과 유사하므로 다시 설명하지 않고 변경, 추가된 내용만 기술한다.

3.1 X-tree의 노드 구조의 확장

XML 문서에서 ID속성을 갖는 엘리먼트의 경우, 유일성을 제공하는 ID속성을 X-tree Diff에서는 충분히 활용하지 못했다. 따라서 X-tree Diff+에서는 X-tree 노드에 ID속성을 위해 iMD 필드를 추가하고 동일 ID속성을 갖는 노드의 쌍

의 대응에 사용한다. iMD 필드에는 $string(MD(Label(\tau_i) \cup$ 'ID 애트리뷰트 값'))이 초기화 과정에서 저장된다.

3.2 기존 알고리즘들의 분석 및 복사연산의 도입

X-tree Diff+에서는 X-tree Diff와 XyDiff 알고리즘과는 달리 복사연산을 지원한다. 복사연산은 동일 tMD 값을 갖는 서브트리가 많을 때 유용하다. 복사연산의 정의는 다음과 같다.

$Copy(\tau_i, \tau_j, k)$: 복사연산으로 서브트리 $ST(\tau_i)$를 노드 τ_j 의 k번째 자식노드로 복사한다.

동일한 서브트리의 대응에 대해 기존 두 알고리즘을 비교해보자. X-tree Diff에서는 1 단계에서 일대일 관계의 동일 서브트리들 간에 이루어지며, 3 단계에서 대응의 하향 확대 과정에서 대응된 부모노드들의 자식노드들의 tMD가 같은 경우에 허용한다. 반면, XyDiff에서는 2 단계의 대응의 하향 확대 과정에서 대응된 노드로부터 같은 깊이(depth)에 있는 자손노드들 중 동일한 SubtreeHash를 갖는 노드를 찾아 이를 루트로 하는 서브트리들을 후위 순회 순서에 따라 대응시킨다. 즉, X-tree Diff는 XyDiff에 비해 서브트리의 대응 대상이 제한된다. 따라서 원본 트리와 대상 트리에서 대응이 결정되지 않은 노드들이 상대적으로 늘어나게 되어 삽입, 삭제연산이 좀 더 빈번히 발생되는 경향이 있다.

(그림 8)은 이를 예시한다. (그림 8)에서 노드 τ_{100}, τ_{200}, τ'_{100}, τ'_{200}은 모두 서로 다른 Label 값을 갖고, 노드 τ_{10}, τ'_{110}, τ'_{120}, τ'_{130} 그리고 τ'_{140}은 모두 같은 tMD(또는 subtreehash) 값을 갖고 있으며 현재 $\tau_{1000} \rightarrow \tau'_{1000}$의 대응이 존재한다고 가정하자. 이 때 X-tree Diff의 경우에는 1 단계에서는 모호한 대응의 연기 규칙으로 인해 아무런 대응이 생성되지 않으며, 3 단계에서도 역시 서브트리의 대응이 생성되지 않는다. 반면, XyDiff의 경우는 $ST(\tau_{10}) \rightarrow ST(\tau'_{110})$의 대응이 생성되고 서브트리 $ST(\tau'_{120})$, $ST(\tau'_{130})$, 그리고 $ST(\tau'_{140})$은 대응되지 않은 채로 남는다.

복사연산의 도입은 위의 두 알고리즘들의 이러한 문제점들을 보완할 수 있다. 마지막 단계에서 대응되지 않은 서브트리들 중에서 동일한 tMD를 갖는 서브트리들을 순서에 의한 대응이 가능하다. X-tree Diff+의 경우, 트리 τ의 서브트리와 트리 τ'의 첫 번째 동일 서브트리에 대해서는 이동연산으로 대응시키고, τ'의 나머지 서브트리들에 대해서는 복사연산을 적용한다. (그림 8)에서 파선 화살표로 표시된 τ_{10}과 τ'_{110}은 이동연산으로 대응시키고, τ_{10}과 τ'_{120}, τ_{10}과 τ'_{130} 그리고 τ_{10}과 τ'_{140}는 복사연산으로 대응시킨다. 자세한 설명은 3.3절을 참고하시오.

3.3 X-tree Diff+ 알고리즘의 소개

X-tree Diff+ 알고리즘은 X-tree Diff를 확장한 것으로 본 논문에서는 일부 수정된 단계만을 설명한다. 앞에서와 마찬가지로 X-tree의 생성과 자료구조의 초기화 단계는 생략한다. 자세한 내용은 [2]을 참조하시오.

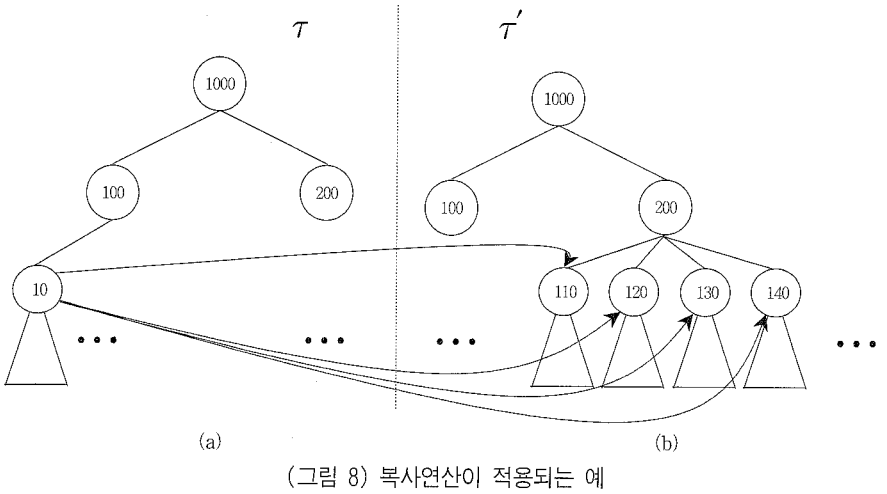
1 단계 - ID속성을 이용한 대응, 유일한 동일 서브트리의 대응

본 단계에서는 기존의 X-tree Diff에서의 1 단계에 XyDiff의 ID속성을 이용하여 대응을 찾아내는 작업을 추가한 것이다.

2 단계, 3 단계 : 각각 X-tree Diff 2, 3 단계와 동일.

4 단계 - 대응의 튜닝(Tuning)

본 단계에서는 노드의 대응 효율성에 대한 분석을 기반으로 일부 비효율적인 대응들에 대해 튜닝 작업을 수행한다. 노드의 대응 효율성은 현 노드와 자식 노드들의 대응의 일관성을 측정하는 것으로 자식노드들의 긍정적 대응의 수 $P\#$(number of positive matches)와 부정적 대응의 수 $N\#$(number of negative matches)에 기초하여 정의된다. 우선 임의의 노드 τ_i에 대한 긍정적 대응의 수 $P\#$와 부정적 대응의 수 $N\#$을, 그리고 노드 τ_i의 대응 효율성 $E(\tau_i)$(efficiency of matching) 다음과



(그림 8) 복사연산이 적용되는 예

같이 정의한다.

$P\#(\tau_i)$ = $(P(M(C_k(\tau_i))) = M(\tau_i))$를 만족하는 τ_i의 자식 노드 C_k의 수

$N\#(\tau_i)$ = $(P(M(C_k(\tau_i))) \neq M(\tau_i))$를 만족하는 τ_i의 자식 노드 C_k의 수

$E(\tau_i)$ = 긍정적 대응된 자식노드 수/긍정적+부정적 대응된 자식노드 수

= $P\#(\tau_i)/(P\#(\tau_i)+N\#(\tau_i))$

즉 $P\#(\tau_i)$는 τ_i의 자식노드들 중 자식노드의 대응 노드, 그 대응 노드의 부모노드와 τ_i의 대응 노드가 같은 τ_i의 자식노드의 수를 의미한다. 즉 이는 부모노드와 일관된 대응 구조를 갖는 자식노드들의 수를 나타낸다. 반면 $N\#(\tau_i)$는 정반대의 경우로 대응이 결정된 자식노드들 중 τ_i와 일관된 대응구조를 갖지 않는 자식노드들의 수를 나타낸다. 노드 τ_i의 대응 효율성 $E(\tau_i)$는 대응이 결정된 τ_i의 자식노드들에 대한 τ_i의 대응과 일관된 대응을 하는 자식노드들의 비율을 의미한다.

한편, 일부 대응 효율성이 낮은 노드에 대한 대응을 튜닝하기 위해서는 노드에 대한 대응 가능한 노드들에 대한 정보가 필요하다. 이를 위해, 임의 노드 τ_i에 대한 대응 가능한 노드들의 리스트 lAM(list of alternative matches)을 다음과 같이 정의한다.

$lAM(\tau_i)$ = {<τ'_j, child C_k의 수> | $Label(\tau'_j)=Label(\tau'_j)$ $\wedge$ $\tau'_j \neq M(\tau_i)$ $\wedge$ $\tau'_j=P(M(C_k(\tau_i))$ for some child $C_k(\tau_i)$}

$lAM(\tau_i)$는 τ_i의 자식노드들 중 이의 대응 노드의 부모노드로, τ_i와 Label이 같아 대응 가능한 노드 τ'_j와 τ'_j와의 대응을 가능하게 하는 τ_i의 자식노드의 수로 구성되는 튜플들의 집합이다. 이 때 대응을 가능하게 하는 자식노드들의 수를 자식노드들에 의한 대응호감지수라 한다. 즉, $lAM(\tau_i)$은 τ_i의 자식노드들의 대응확대의 시각에서 본 대응 가능한 노드들의 정보라고 할 수 있다. 한편, $lAM(\tau_i)$ 중에서 자식노드들의 대응호감지수가 가장 큰 노드를 τ_i에 대한 가장 대응 가능성이 높은 노드로 $fN(\tau_i)$(fittest node) 그리고 그 대응호감지수를 $fV(\tau_i)$(fittest value)라고 한다.

대응의 튜닝 작업은 X-tree τ을 후위 순회로 방문하면서, 방문한 노드 τ_i의 $E(\tau_i)$가 0.5이상이면 다음 노드로 진행한다. 그러나 $E(\tau_i)$)가 0.5미만이면 $lAM(\tau_i)$로부터 $fN(\tau_i)$와 $fV(\tau_i)$를 구하고 $fN(\tau_i)$의 현 대응 노드 τ_k(즉, $\tau_k=M(fN(\tau_i))$)에 대해 $fV(\tau_i) > P\#(\tau_i) + fN(\tau_k$일 때 τ_i을 $fN(\tau_i)$로 대응시키고 τ_k를 $M(\tau_i)$로 대응시킨다.

(그림 9)는 튜닝의 예를 보인다. 파선은 이미 대응이 결정된 노드로 현재의 대응은 $\tau_{11}\rightarrow\tau'_{17}$, $\tau_{12}\rightarrow\tau'_{21}$, $\tau_{13}\rightarrow\tau'_{22}$, $\tau_{14}\rightarrow\tau'_{23}$, $\tau_{24}\rightarrow\tau'_{24}$, $\tau_{15}\rightarrow\tau'_{18}$ 그리고 $\tau_{25}\rightarrow\tau'_{25}$이다. X-tree Diff+는 튜닝에 필요한 $P\#(\tau_i)$, $N\#(\tau_i)$, $E(\tau_i)$ 그리고 $lAM(\tau_i)$ 자료구조를 생성하는데, (그림 9)에서 튜닝의 대상이 되는 노드는 τ_{15}와 τ_{25}이며 이들 노드에 대한 튜닝 자료는 (그림 9)에서 보이고 있다.

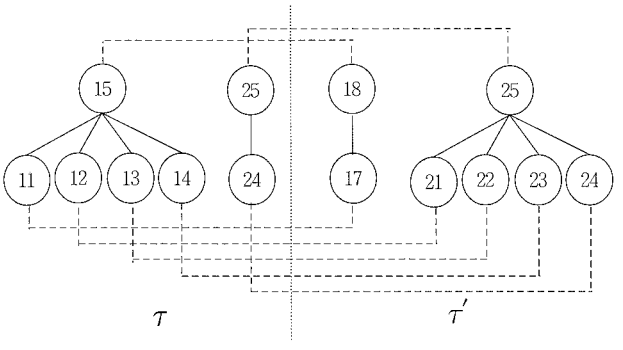
X-tree τ을 후위 순회로 방문하면서, 방문한 노드 τ_{15}의 $E(\tau_{15})$가 0.5미만이 되므로 $lAM(\tau_{15})$로부터 $fN(\tau_{15}) = \tau'_{25}$와 $fV(\tau_{15}) = 3$을 구하고 $fN(\tau_{15})$의 현 대응 노드 τ_{25}에 대해 $fV(\tau_{15}) > P\#(\tau_{15}) + P\#(\tau_{25})$이므로 τ_{15}를 $fN(\tau_{15})$인 노드 τ'_{25}와 대응시키고 τ_{25}는 $M(\tau_{15})$인 노드 τ'_{18}과 대응시킨다.

이러한 튜닝과정을 적용하면, 2.3절에서 언급한 문제점들은 발생하지 않는다.

5 단계 - 대응되지 않은 동일 서브트리의 대응

1-4 단계의 대응 과정 후에도 X-tree τ와 τ'에는 대응이 결정되지 않은 노드들이 있을 수 있다. 이들 중 일대다, 다대일 그리고 다대다의 관계의 동일 서브트리들 중 대응이 이루어지지 않은 서브트리들에 대해서는 이동연산이나 복사연산을 통해 추가적인 대응을 결정할 수 있다.

이를 위해서는 X-tree τ와 τ'를 각각 너비우선으로 순회



τ_{15}의 튜닝 자료

$P\#(\tau_{15})$	1
$N\#(\tau_{15})$	3
$E(\tau_{15})$	1/4 = 0.25

τ_{25}의 튜닝 자료

$P\#(\tau_{25})$	1
$N\#(\tau_{25})$	0
$E(\tau_{25})$	1/1 = 1.00

(그림 9) 대응 튜닝의 예

하면서 대응이 결정되지 않은 노드들을 tMD값을 키로 하는 해시테이블들을 만들어 동일 tMD 값을 갖는 노드들을 서로 대응시킨다. 이 때 (그림 10)의 해시테이블이 사용되는데, 이 해시테이블의 엔트리는 tMD값과 노드 리스트의 포인터로 구성되어 동일한 tMD 값을 갖는 노드들의 리스트(linked list)를 저장한다. τ를 너비우선으로 순회하면서 생성되는 해시테이블을 S_Htable, τ'로부터의 해시테이블을 T_Htable라고 하자.

동일 서브트리의 대응은 다음과 같이 진행된다.

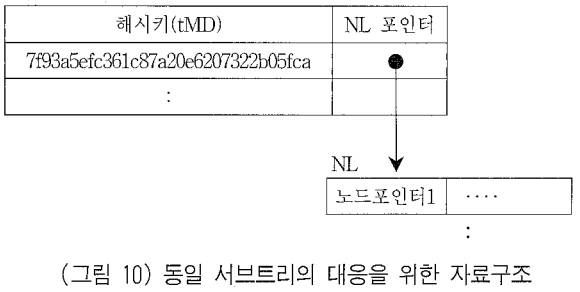
(1) T_Htable에 등록된 엔트리들을 순서대로 방문하면서, 방문한 임의 엔트리 (h_key_p, T_NL_p)의 해시키 h_key_p로 S_Htable를 룩업한다. 룩업에 실패하면 T_Htable의 다음 엔트리에 대해 시도한다. (2) 룩업에 성공했을 때, 룩업된 S_Htable의 엔트리의 NL 필드를 S_NL_q라고 하면 노드 리스트 T_NL_p와 S_NL_q에서 동일 순서의 노드의 쌍(τ_i, τ'_j)을 얻어 $ST(\tau_i)$와 $ST(\tau'_j)$를 대응시킨다. 리스트 S_NL_q에 있는 노드의 수를 m, T_NL_p에 있는 노드의 수를 n이라 할 때, $m <= n$일 경우, 처음부터 m개까지의 노드의 쌍을 이동연산으로 동일 서브트리 대응시키고, T_NL_p의 나머지 $n-m$개의 노드에 대해서는 S_NL_q의 마지막 노드 즉 이를 루트로 하는 서브트리에 대한 복사로 처리한다. 한편, $m > n$일 경우는 n만큼 이동연산으로 동일 서브트리 대응시키며, S_NL_q에 남은 서브트리는 대응되지 않은 상태로 둔다.

4. 성능 평가

본 절에서는 X-tree Diff+의 시간 비용을 분석하고 X-tree Diff와 XyDiff와의 성능 비교에 관한 실험결과를 보인다.

우선 X-tree Diff+의 단계별 시간 비용을 분석한다. $|\tau|$는 X-tree τ의 노드의 개수를 나타내며, n은 X-tree τ와 τ'의 해당 XML 문서의 노드 수의 합을 의미한다. X-tree Diff+는 X-tree Diff 알고리즘을 확장한 것으로 본 논문에서는 X-tree Diff에서 확장된 부분에 대한 비용 분석만 보인다. X-tree Diff+에서 신·구문서에 대한 X-tree의 생성과 알고리즘에 사용될 자료구조의 초기화 비용은 X-tree Diff에서와 마찬가지로 $O(n)$이다.

1 단계에서는 ID속성을 이용하여 노드 대응을 결정하는 부분이 추가되었다. ID속성을 키로 하는 해시테이블을 통해 검색하면 모든 노드가 ID속성을 갖는 경우, 해시테이블의 엔트리 수는 $|\tau|$이 되므로 ID 애트리뷰트를 통한 노드의 대응비용은 $O(|\tau|)$, 즉 $O(n)$에 바운드된다.



(그림 10) 동일 서브트리의 대응을 위한 자료구조

2, 3단계는 X-tree Diff와 동일하여 생략한다.

4 단계는 이전 단계에서 생성된 비효율적인 대응들을 튜닝 하는 과정으로 튜닝을 위한 자료구조 생성 과정과 튜닝 과정으로 구성된다. 자료구조 생성과정은 X-tree τ에 대한 한 번의 순회로 이루어지며 비용 $O(|\tau|)$이 소요된다. 튜닝 과정 또한 X-tree τ에 대한 한 번의 순회로 이루어진다. 3.3 절에서는 이해를 돕기 위해 $lAM(\tau_i)$를 리스트 구조로, $lAM(\tau_i)$ 중에서 $fN(\tau_i)$을 구하는 것으로 설명하였으나 구현 시에는 $lAM(\tau_i)$을 해시테이블로, $fN(\tau_i)$은 별도의 자료구조로 관리된다. 따라서 4 단계의 전체 비용은 $O(|\tau|)$이 된다.

5 단계는 대응이 결정되어있지 않은 동일 서브트리를 대응시키는 과정이다. 대응이 결정되지 않은 노드들의 tMD 값을 키로 하는 해시테이블 생성은 한 번의 τ와 τ'의 순회로 충분하다. 한편, 서브트리의 대응을 찾는 과정에서 해시테이블과 리스트 자료구조가 사용되는데, 자료구조의 전체 엔트리 수는 $|\tau|+|\tau'|$이하가 된다. 그러므로 (그림 10)의 자료구조 T_Htable에 등록된 엔트리들을 순차적으로 검색하며 추가적인 이동 및 복사연산으로 대응을 찾는 비용은 $O(|\tau|+|\tau'|)$에 바운드된다. 5 단계의 전체 비용은 $O(|\tau|+|\tau'|)$이 된다.

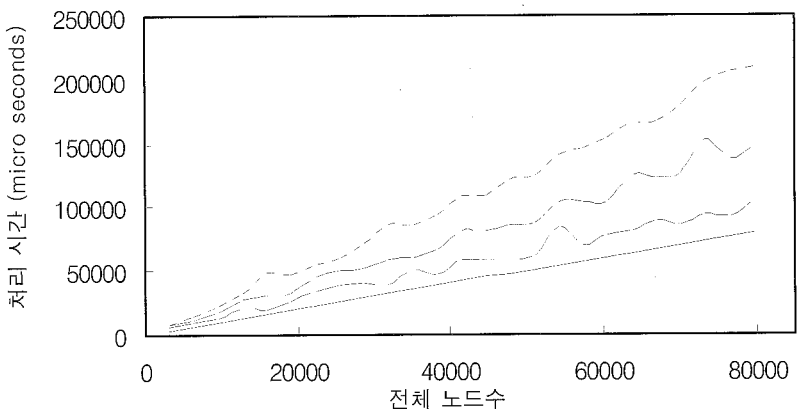
따라서 X-tree Diff+의 전 과정에 대한 시간 비용은 $O(n)$이 되며, 이는 X-tree Diff와 XyDiff[2]의 시간 비용과 같다.

(그림 11)는 전체 노드 수에 따라 X-tree Diff, X-tree Diff+ 그리고 XyDiff가 대응을 찾는데 소요된 시간을 측정한 실험 결과를 보이고 있다. 세로축의 처리 시간은 대응을 찾는 각 단계에서 소요된 시간의 합을 나타내며 알고리즘의 자료구조 생성 및 초기화 단계와 편집스크립트 생성 단계에서의 소요 시간은 포함되지 않았다. (그림 11)에서 X-tree Diff와 XyDiff 그리고 X-tree Diff+ 모두 전체 노드수의 증가함수와 같이 선형 기울기를 보여주고 있어 알고리즘의 시간 비용이 $O(n)$에 바운드됨을 보이고 있다. 소요시간의 효율성 측면에서는 X-tree Diff, X-tree Diff+, 그리고 XyDiff 순으로 되어있음을 알 수 있다.

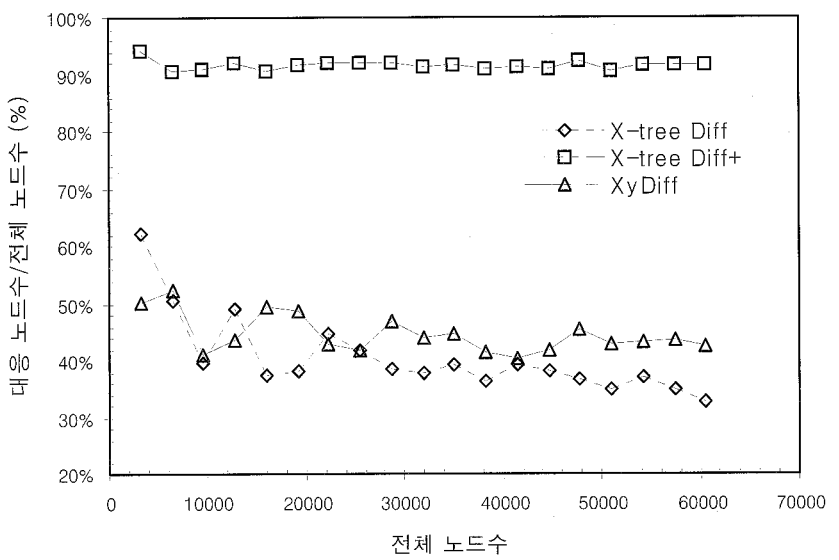
구현은 모두 MS VC++ 7.0을 통해 이루어졌으며, 테스트 데이터는 Niagara 프로젝트[12]에서 사용한 XML 문서생성기(Synthetic XML Data Generator)[3]를 이용하여 생성하였다. 실험은 CPU 600MHz, 메모리 256MB인 PC를 통해 이루어졌다.

(그림 12)는 전체 노드 수에 대한 각 알고리즘이 생성한 대응 노드 수의 비율을 통해 대응의 질(quality)를 비교한다. 세로축은 전체노드에 대한 대응 노드수의 비(대응률)를 퍼센트로 표기하였다. (그림 12)에서 볼 수 있듯이 X-tree Diff는 XyDiff보다 다소 대응률이 떨어진다. 이는 X-tree Diff가 빠

2) 초기 XyDiff는 비-대응 서브트리 중 가장 큰 서브트리의 대응을 먼저 찾아 대응시키는 전략을 사용하였다. 가장 큰 서브트리를 얻기 위해 가중치로 서브트리를 정렬한 순서 큐(ordered queue)를 활용하였고 이로 인해 $O(nlog(n))$의 시간 비용을 보였다. 최근 버전에서는 순서 큐를 사용하지 않으므로 $O(n)$의 시간 비용으로 처리된다.



(그림 11) 전체 노드수에 따른 대응 처리 시간 측정 실험 결과



(그림 12) 전체 노드수에 따른 대응률

르나 동일 서브트리 대응에 X-tree Diff가 XyDiff보다 강한 제약조건을 사용하기 때문이다. 반면, X-tree Diff+는 대응률에 있어 다른 두 알고리즘에 비해 월등하게 뛰어나다. 이는 1 단계의 ID속성을 통한 대응, 4 단계의 튜닝 작업, 그리고 5 단계에서 동일 서브트리 대응을 추가함으로 대응의 질이 상당히 개선되었음을 의미한다. 즉, 일부 노드의 부적절한 대응으로 불필요하게 유발된 삽입 혹은 삭제 연산들이 대폭 감소되었음을 나타내는 것이다.

5. 결론

본 논문에서는 기존의 휴리스틱 변화 탐지 알고리즘 XyDiff와 X-tree Diff을 분석하여 이들 알고리즘의 문제점들을 제시하고 이들을 문제점들을 개선한 X-tree Diff+ 알고리즘을 제안하였다.

X-tree Diff는 알고리즘이 간결하여 실행 속도가 XyDiff에 비해 빠르나 알고리즘이 생성하는 대응의 질은 XyDiff에 비해 약간 떨어지는 경향이 있다. 이는 XyDiff의 경우, 알고리즘의 구성이 복잡하고 트리의 순회 빈도도 높아 실행 속도가 떨어지나, 대응의 확대 과정에서 노드의 가중치를 사용하고 동일 서브트리의 대응 시 조상 노드의 대응관계를 고려하여 대응의 질을 높이고 있기 때문이다. 반면, X-tree Diff는 실시간의 변화 탐지 응용분야에서는 최소비용의 편집스크립트의 생성보다는 빠른 알고리즘이 필요함을 인지하여 설계된 휴리스틱 알고리즘으로 설계원칙이 알고리즘의 명확성과 간결함에 있었다. 이로 인해 실행속도가 빠르나 대응의 질이 다소 떨어지는 한계가 있다.

본 논문에서 제안한 X-tree Diff+는 X-tree Diff 정도의 실행 효율을 유지하면서 대응의 질을 높이고자 설계된 알고리즘이다. 이를 위해 ID속성 대응 과정을 추가하고 대응의 튜닝 작업과 복사연산을 추가하였다. X-tree Diff+의 시간복잡도가 $O(n)$(단, n은 노드 수)이며 실행속도 측면에서는 XyDiff 보다 뛰어나고 X-tree Diff 보다 약간 떨어지나 대응률로 표현되는 대응의 질은 훨씬 개선된 결과를 보이고 있다. 실험 결과에서 알 수 있듯이 대응률로 표현된 대응의

질은 XyDiff 보다도 훨씬 뛰어남을 알 수 있다.

뛰어난 실행성능과 개선된 대응의 질로 인해 X-tree Diff+ 알고리즘은 XyDiff와 X-tree Diff 알고리즘이 사용되었던 모든 응용분야에 사용될 수 있을 것을 판단된다. 앞으로는 최소비용을 제공하는 알고리즘과의 대응의 질을 비교하는 작업을 통해 X-tree Diff+ 알고리즘의 다양한 특성들을 분석하고 개선 방향을 모색하려고 한다. 또한 X-tree Diff+ 기반의 버전 관리, 해킹 탐지 시스템을 개발하려고 한다.

참 고 문 헌

[1] 김동아, 이석균,"X-tree Diff : 트리 기반 데이터를 위한 효율적인 변화 탐지 알고리즘," 정보처리학회논문지(C), 제10호 제6권, pp.683-694, 2003.

[2] 김동아, "XML 문서에 대한 변화 탐지 및 관리," 단국대학교 전산통계학과 박사학위논문, pp.1-111, 2005.

[3] A. Aboulnaga, J. F. Naughton, and C. Zhang."Generating Synthetic Complex-structured XML Data." In Proceedings of the Fourth International Workshop on the Web and Databases, WebDB, 2001.

[4] "Concurrent Versions System(CVS)," *Free Software Foundation*, http://www.gnu.org/manual/cvs-1.9.

[5] Curbera and D. A. Epstein,"Fast Difference and Update of XML Documents," XTech '99, San Jose, March 1999.

[6] D. A. Kim and S. K. Lee, "Efficient Change Detection in Tree-Structured Data," In Human.Society@Internet 2003, pp.675-681, 2003.

[7] D.T. Barnard, G. Clarke and N. Duncan,"Tree-to-tree correction for document trees," Technical Report, Department of Computing and Information Science Queen's University Kingston Ontario, Canada, January 1995.

[8] E. W. Myers,"An O(ND) Difference Algorithm and Its Variations," *Algorithmica*, 1(2), pp.251-266, 1986.

[9] G. Cobéna, S. Abiteboul and A. Marian,"Detecting Changes in XML Documents," The 18th ICDE, 2002.

[10] K. Tai,"The tree-to-tree correction problem," Journal of the ACM, 26(3), pp.422-433, July 1979.

[11] K. Zhang and D. Shasha,"Simple fast algorithms for the editing distance between trees and related problems," SIAM Journal of Computing, 18(6), pp.1245-1262, 1989.

[12] NIAGARA Query Engine, http://www.cs.wisc.edu/niagara/ .

[13] R. Rivest,"The MD4 Message-Digest Algorithm," MIT and RSA Data Security, Inc., April 1992.

[14] R. Wagner and M. Fischer,"The string-to-string correction problem," *Journal of the ACM*, 21, pp.168-173, 19

[15] S. Chawathe and H. G. Molina,"Meaningful Change Detection in Structured Data," In SIGMOD '97, pp.26-37, 1997.

[16] S. Lu,"A tree-to-tree distance and its application to cluster analysis," IEEE TPAMI, 1(2), pp.219-224, 1979.

[17] S. M. Selkow,"The tree-to-tree editing problem," Information Processing Letters, 6, pp.184-186, 1977.

[18] XyDiff Tools, http://pauillac.inria.fr/cdrom/www/xydiff/index-eng.htm.

[19] Xyleme Project, http://www.xyleme.com/en/

[20] Y. Wang, D. Dewitt and J. Cai,"X-Diff: An effective change detection algorithm for XML Documents," in 19th ICDE, India, March 2003 .

이 석 균

e-mail : sklee@dankook..ac.kr

1982년 서울대학교 경제학과 졸업(학사)

1990년 University of Iowa 전산학(석사)

1993년 University of Iowa 전산학(박사)

1993년 - 1997년 세종대학교 정보처리학과 전임강사

1997년 ~ 현 재 단국대학교 컴퓨터과학전공 교수

관심분야 : 데이타 모델링, 데이타베이스에서 불완전 정보관리, 객체지향 데이타베이스 시스템, 데이타베이스 질의어, 시각질의어, 다중처리기에서의 실시간 스케줄링, 데이터웨어하우스, 데이터마이닝, 데이터베이스 시각 질의어, 문서의 Diff 알고리즘, XML, Change Detection, Version Control 등

김 동 아

e-mail : dakim70@dankook.ac.kr

1993년 서울교육대학(학사)

1999년 단국대학교 교육대학원 전산교육과(석사)

2004년 단국대학교 대학원 전산통계학과 전산학전공(박사)

2000년 ~ 2002년 (주)소프트웨어컨설팅그룹 선임연구원

2003년 ~ 현재 화랑초등학교 교사

2006년 ~ 현재 단국대학교 컴퓨터과학전공 겸임교수

관심분야 : 데이타 모델링, 객체지향 데이타베이스 시스템, 데이터웨어하우스, 데이터마이닝, 컴퓨터 교육, XML Change Detection, Version Control 등