X-tree Diff : 트리 기반 데이터를 위한 효율적인 변화 탐지 알고리즘

이 석 균[†] · 김 동 아[††]

요 약

인터넷 사용이 급속도로 증가함에 따라 XML/HTML 문서와 같이 트리 구조로 표현되는 데이터의 변화 탐지가 중요한 연구 분야로 등장하고 있다. 본 논문에서는 효율적인 변화 탐지를 위한 데이터 구조로 X-tree와 이에 기초한 휴리스틱 변화 탐지 알고리즘 X-tree Diff를 제안한다. X-tree Diff는 X-tree의 변화 내용에 대한 최소 비용 편집 스크립트를 찾는 알고리즘과는 달리 휴리스틱 트리 대응 알고리즘을 통해 변화 내용을 확인하는 알고리즘으로, X-tree에 속한 모든 노드에 대해 각각의 노드를 루트로 하는 서브트리의 구조와 서브트리에 속한 노드들의 데이터들을 128비트 해시값으로 표현한 값인 트리 MD를 각 노드에 저장하고, 이를 변화 탐지 과정에서 활용하여 신-구 버전의 X-tree들에 속한 서브트리들의 비교를 효율적으로 처리한다. X-tree Diff는 4 단계로 구성되며 ① 신-구 버전의 X-tree 노드들에 대해, 우선 1 : 1 대응이 가능한 모든 동등한 서브트리 쌍을 찾고, ② 이들 서브트리 쌍의 루트로부터 신-구 버전의 X-tree의 루트까지의 경로 상에 존재하는 노드들에 대한 대응관계를 결정한다. ③ 그 후 신-구 버전의 X-tree의 루트들로부터 깊이 우선 탐색으로 노드를 방문하며 대응이 결정되지 않은 노드들에 대한 대응 여부를 결정해나간다. ④ 마지막까지 대응여부가 결정되지 않은 노드들은 삭제나 삽입된 것으로 간주한다. X-tree Diff는 XML 문서들에 대한 버전닝(Versioning)을 목적으로 설계된 BULD Diff 알고리즘과 달리 XML/HTML에 공통적으로 사용할 수 있을 뿐 아니라, 알고리즘이 명확하고 간결하여 다양한 형태의 확장이 가능하다. 알고리즘의 성능도 개선되어 신-구 X-tree의 노드의 수를 n이라 할 때, $O(n)$의 시간 복잡도를 갖는다. 제안된 알고리즘은 현재 보안 관련 상용 시스템인 WIDS(Web-Document Intrusion Detection System)에서 사용되고 있으며, 본 논문에서는 WIDS를 이용하여 20여개 신문-방송 사이트에서 변화가 탐지된 11,000개 페이지에 대한 성능평가를 보이고 있다.

X-tree Diff : An Efficient Change Detection Algorithm for Tree-structured Data

Suk Kyoon Lee[†] · Dong Ah Kim[††]

ABSTRACT

We present *X-tree Diff*, a change detection algorithm for tree-structured data. Our work is motivated by the need to monitor massive volume of web documents and detect suspicious changes, called defacement attack on web sites. From this context, our algorithm should be very efficient in speed and use of memory space. X-tree Diff uses a special ordered labeled tree, X-tree, to represent XML/HTML documents. X-tree nodes have a special field, tMD, which stores a 128-bit hash value representing the structure and data of subtrees, so that it compares between subtrees efficiently. X-tree Diff, supporting four basic operations (insert, delete, update, and move), uses tMD fields to match identical subtrees from the old and new versions. During this process, X-tree Diff uses *the Rule of Delaying Ambiguous Matchings*, implying that it perform exact matchings where a node in the old version has one-to-one correspondence with the corresponding node in the new, by delaying all the others. It drastically reduces the possibility of wrong matchings. X-tree Diff propagates such exact matchings upwards in Step 2, and obtain more matchings downwards from roots in Step 3. In Step 4, nodes to be inserted or deleted are decided. We also show that X-tree Diff runs in $O(n)$, where n is the number of nodes in X-trees, in worst case as well as in average case. This result is even better than that of BULD Diff algorithm, which is $O(n \log(n))$ in worst case. We experimented X-tree Diff on real data, which are about 11,000 home pages from about 20 web sites, instead of synthetic documents manipulated for experimentation. Currently, X-tree Diff algorithm is being used in a commercial hacking detection system, called the WIDS (Web-Document Intrusion Detection System), which is to find changes occurred in registered websites, and report suspicious changes to users.

키워드 : 변화 탐지(Change Detection), Diff, 트리 구조(Tree Structured Data), 대응(Matching)

1. Introduction

Nowadays, most information is being provided through WWW. As the need to access up-to-date information at right moment has been drastically increased, the technology of detecting changes occurring in web documents in real-time becomes indispensable. For example, stock analysts may want to access news and economic articles affecting stock markets as soon as they are released. Salesmen may be interested in monitoring prices of products sold by competitors,

※ The present research was conducted by the research fund of Dankook University in 2002.
† 정 회 원 : 단국대학교 컴퓨터과학과 교수
†† 정 회 원 : 화랑초등학교 교사
논문접수 : 2003년 7월 4일, 심사완료 : 2003년 9월 19일

and so on. Change detection technology is also useful for areas of version management [1, 2], data warehouse [3], and active databases [4].

Even though the needs for effective change detection technologies are increasing, researches on these technologies for tree-structured data are far from satisfactory. In this paper, we propose X-tree Diff, heuristic algorithm for detecting changes in tree-structured data, which is $O(n)$ in time complexity, where is n the number of nodes.

1.1 Background

The problem on change detection has been studied since 1970s, and early works on this problem has focused on generating the minimal cost edit operations based on the distance between text files [5, 6]. R.Wagner et. al. proposed the 'compare' algorithm [23] which, based on DAG, produce the minimal cost edit script for two strings x, y. However, the cost is proportional to $O(|x|\times|y|)$, where $|x|$ means the length of string x. The famous program for change detection in plain text files may be GNU *diff* utility, which has been implemented based on LCS (Longest Common Subsequence) algorithm [5], and is being used in *CVS* utility [6], the version management system for programs.

Researches on change detection for tree-structured data have been carried out since late 1970s [8-17]. Early works focused on the computation of the minimal cost edit script for tree-structured data [8-10]. Selkow's algorithm considered trees of depth two, where insertion and deletion can occurs only in leaf nodes [8]. Tai presented an algorithm producing the minimal cost edit script for ordered labeled trees, which is known as the first non-exponential algorithm [9].

Recently, as uses of tree-structured data such XML/HTML documents are increasing rapidly, due to the explosive growth of uses of internet, the problem on the change detection for tree-structured data draws much attention from research community [11-17]. The general problem on change detection for tree-structured data is known as *NP*-hard [12]. So, many recent works proposed heuristic solutions instead of optimal ones.

Zhang and Shasha proposed a fast algorithm to find the minimum cost editing distance between two ordered labeled trees [18]. Given two ordered trees T_1 and T_2, their algorithm finds a minimal edit script in time $O(|T_1|\times|T_2|\times min\{depth(T_1), leaves(T_1)\}\times min\{depth(T_2), leaves(T_2)\})$, where $|T|$, depth(T), and leaves(T) represent the number of nodes, the depth, and the number of leaves in T, respectively. Chawathe et. al., also, proposed an efficient algorithm, called MH-Diff. In MH-Diff, the change detection problem is transformed to the problem of computing a minimum-cost edge cover of a bipartite graph [12]. However, the worst case of algorithm is $O(n^3)$ in time complexity, where n is the number of nodes.

To improve the performance, recent works often use hash values to represent the contents and structure of subtrees for the efficient comparison of subtrees [15-17, 19]. XMLTree Diff[1] applies Zhang and Shasha's algorithm [18] to detect changes in XML documents. It also uses hash values of subtrees to prune identical subtrees [15].

BULD Diff[2] algorithm [16], based on both Lu's and Selkow's algorithms [8, 10], is designed for XML data warehouse and versioning. This algorithm uses 32-bit hash values (called a signature) to represent the contents and structure of subtrees, and the notion of weight (i.e., a number of subtree nodes) for matching process. It achieves $O(n\ log\ n)$ complexity in execution time, using hashing and heuristic techniques.

X-Diff+[3] [17], based on the unordered tree model, also uses modified DOMHash [19] to represent the contents and structure of subtrees. Using the insert, delete, and update operations, and their cost functions, X-Diff+ computes the edit distance between two documents, and produce reasonable good edit scripts with time cost of $O(|T_1|\times|T_2|\times max\{\deg(T_1), \deg(T_2)\}\times log_2\{\deg(T_1), \deg(T_2)\})$ in worst case, where deg(T) represents the maximum out-degree in tree T.

1.2 Motivations and Characteristics of X-tree Diff

One of hot issues on internet security is on defacing web sites. In these days, it is not strange to hear about hackers' defacement attacks to major web sites. One of solutions to this problem is to monitor home pages of web sites to see if defacement attacks occur, and notify attacks to operators, if suspicious changes detected, in order for them to cope with the situation properly.

This work is motivated by the need to monitor a huge amount of HTML documents, and detect changes in those documents in real-time. Change detection algorithms to be

1) Available at http://www.alphaworks.ibm.com/tech/xmltreediff.
2) Available at http://www-rocq.inria.fr/~cobena/cdrom/www/xydiff/eng.htm.
3) Available at http://www.cs.wisc.edu/~yuanwang/xdiff.html.

used in such systems should satisfy the following properties : efficiency, quality and extensibility. These algorithms should be fast enough to handle malign defacement attacks in real-time, and also produce reasonable good quality of output. The algorithms may be extensible in various ways, in order to meet needs from different applications. For example, users may want to have intelligent change detection systems capable of distinguishing malign changes from routine changes, and so on.

However, most existing algorithms are not fast enough. BULD Diff algorithm, which is known as fastest among existing algorithms, runs in time $O(n \log n)$ in worst case. Also, BULD Diff is so complex that it is not easy to understand and is difficult to extend. It is claimed that the output is reasonable close to "optimal." But, according to a recent report, the quality of the output in some cases is not good enough [17].

In this paper, we propose X-tree Diff, the algorithm designed to detect changes in tree-structured data such as XML/HTML documents. In X-tree Diff, we use X-tree, similar to the structure of the Document Object Model (DOM) [22], to represent XML/HTML documents. X-tree are labeled ordered trees, designed to easily derive the distance [4] between two tree-structured documents.

X-tree Diff is fast, reliable, simple and clear change detection algorithm. X-tree Diff runs fast in time cost $O(n)$ in worst case, which is even faster than BULD Diff algorithm. We use 128-bit hash values, in matching nodes, which are generated from the Message Digest algorithm (MD4) [20], known as balanced and reliable in computer security area. Because of using MD4 as hash function in X-tree Diff, matching process in X-tree Diff is more reliable than any other algorithms using hashing, so that the possibility of wrong matchings may be reduced. In X-tree Diff, we also use *the Rule of Delaying Ambiguous Matchings*. It implies that, during matching process, X-tree Diff performs exact matchings where a node in the old version has one-to-one correspondence with the corresponding node in the new, by delaying all the others. It reduces the possibility of wrong matchings. X-tree Diff propagates such exact matchings upwards in Step 2, and obtain more matchings downwards from roots in Step 3. In Step 4, nodes to be inserted or deleted are decided.

Since August in 2002, X-tree Diff has been being used in a commercial hacking detection system, called the WIDS (Web-Document Intrusion Detection System). About 700 web sites and over 7000 web pages form these web sites are registered to WIDS, and being monitored every 5 to 20 minutes. One last characteristics of X-tree Diff is simplicity. In comparing with other existing algorithms, the structure of X-tree Diff is simple and clear, so that it can be extended easily.

In the rest of the paper, we present the change representation model in Section 2. In Section 3, we describe our X-tree Diff algorithm in detail. We analyze X-tree Diff and are showing experiments in Section 4. The last section a conclusion and future work.

2. The Model for Representing Changes in XML Documents

In this section, we propose the notion of *X-tree*, which represents a XML document as a tree. X-tree is a core data structure used in X-tree Diff algorithm to represent changes in XML documents. We, then, introduce the notions of *Node_MD* and *Tree_MD* in Section 2.2. The notion of Node_MD is used to represent the information stored in a node of X-tree, while the notion of Tree_MD to represent information stored in a subtree of X-tree. In Section 2.3, we define the basic edit operations used to describe the changes between two X-trees.

2.1 X-tree

X-tree is a labeled ordered tree, designed to reflect the hierarchical structure of XML documents. In addition to fields used to maintain the tree structure, a node of X-tree consists of eight fields shown in (Figure 1). Among these fields, *Label*, *Type* and *Value* fields altogether represent an element of XML documents. *Index* field is used to distinguish sibling nodes in X-tree that have the same label, while *nMD*, *tMD*, *nPtr* and *Op* fields to detect changes in X-tree Diff.

Label	Type	Value	Index	nMD	tMD	nPtr	Op

(Figure 1) Logical structure of a node of X-tree.

First, we describe how to represent an XML document in X-tree. The node of X-tree represents either the *element node* or *text node* (PCDATA) in DOM (Document Object Model) [22]. For each text node of an XML document, the

4) The distance may be interpreted as the notion of *delta* in [16].

string "#TEXT" and the text content of the node are stored, respectively, in the Label field and Value field of the corresponding X-tree node. For each element node, the name of the node is stored in the Label field of the corresponding node.

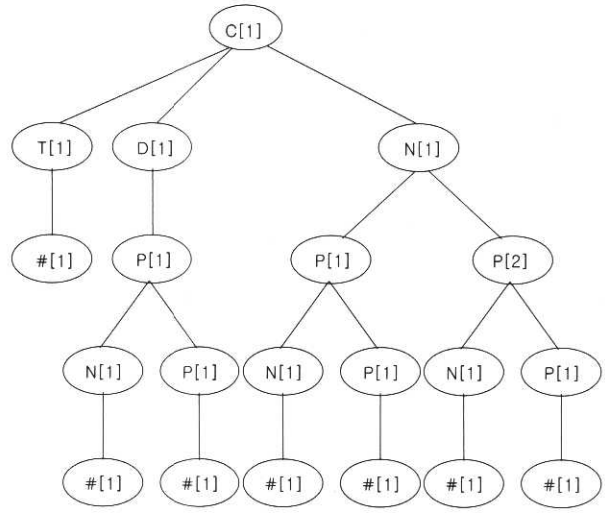
In this paper, for simplicity and clarity of exposition, we decide not to represent the attribute nodes of DOM in X-tree. However, extending our model to include the attribute nodes is not difficult. We can use element nodes in a restricted way to represent attribute nodes[5], or use the Value field to represent the name and contents of all the attributes. We took the second approach for X-tree Diff used in experiments shown in Section 4.

For X-tree sibling nodes whose Label fields have the same value, their Index fields are set to the numbers such as 1, 2, 3, ···, according to the left-to-right order to distinguish these sibling nodes. The number 1 is used as the default value for the others. For the simplicity of writing, we assume that field names also are treated as functions. For example, *Label*(*N*) denotes the value of the Label field of an X-tree node *N*.

```
<Category>
    <Title> LCD Monitor </Title>
    <Discount>
        <Product>
            <Name> 782LE </Name>
            <Price> 875,000 </Price>
        </Product> </Discount>
    <New Products>
        <Product>
            <Name> 577CM </Name>
            <Price> 755,000 </Price></Product>
        <Product>
            <Name> 575LS </Name>
            <Price> 625,000 </Price></Product>
    </New Products>
</Category>
```

(Figure 2) A piece of XML document

Consider an XML document in (Figure 2), which represents a catalog in XML. In (Figure 3), we show a simplified X-tree for the XML document. The label of a node in (Figure 3), which is used to identify a node, comes from the values of the Label and Index fields. However, for the notational simplicity, the first character of the string value of the Label field is used for the labels of nodes. For example, the <Category> element in (Figure 2) is represented by the root node 'C[1]' in (Figure 3), and the second <Product> element of the <New Products> element by the node 'P[2]' that is the second child of 'N[1]'. As seen above, nodes of X-tree can be identified by the value combined from the Label and Index fields. These values are called *ILabel* (indexed label) and defined as *ILabel*(*N*) = *Label*(*N*)[*Index*(*N*)] for an X-tree node *N*.



(Figure 3) An X-tree representation

By the notion of ILabel, we can distinguish all the sibling nodes with a same parent, but fail to uniquely identifying all the individual nodes in X-tree. So, we introduce the notion of *nID* (node identifier) which is defined by extending the notion of ILabel along the tree path. For a node *Np* in an X-tree *T*, the nID value of node *Np* is defined as the concatenation of all the ILabel strings of nodes along the path from the root *Nr* to the node *Np*. Formally it is defined as follows : nID (*Np*) = .Label (*Nr*) [Index (*Nr*)]., ···, .Label (*Np*) [Index (*Np*)]. For example, the nID of the <Category> element in (Figure 2) is '.C[1]', and the nID of the second child <Product> element of the <New Products> element is '.C[1].N[1].P[2]'.

2.2 Node_MD and Tree_MD

Detecting changes in two XML documents requires comparison between two X-trees T_{old} and T_{new}, converted from them. For efficient comparison between X-Trees, X-tree Diff uses the hash algorithm MD4[6] [20, 21] to produce hash values for nodes and subtrees. These hash values are called Node_MDs and Tree_MDs. The Node_MD of a node *N* is

5) BULD Diff [16] and X-Diff+ [17] took the similar approach to represent attributes.

6) MD4 is a reliable hash algorithm for our purpose. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest.

the hash value from applying the MD4 hash function to the contents of the node N. After being computed, the Node_MD is stored in the nMD field of node N. Node_MD is defined as follows.

Node_MD : The Node_MD of a node N is a 32-byte string value, which results from applying MD4 hash function to the concatenation of the values of the Label and Value field of the node N. The Node_MD is defined formally using nMD as follows :

$$\text{nMD}(N) = \text{string}(\text{MD4}(\text{Label}(N) \oplus \text{Value}(N)))$$

where $\oplus$ is a string concatenating operator, and string(h) is a function that receive an 128-bit hash value as the input value, and returns 32-byte hexadecimal character string.

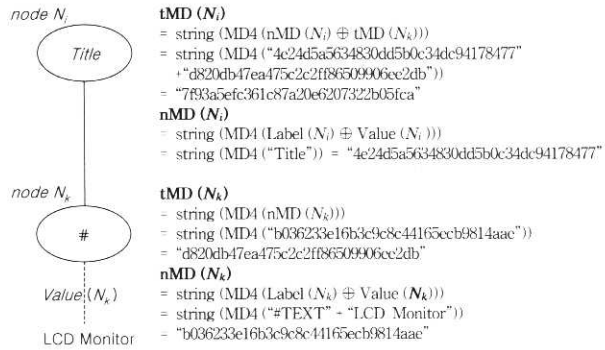
In order to compare two nodes, we simply compare their Node_MDs, instead of the values of the Label, and Value fields [7]. When the Node_MD of a node A is equal to the one of a node B, the nodes A and B are regarded as identical to each other. Tree_MD represents a hash value for the contents of an X-tree. The Tree_MD of an X-tree T is stored in the Field tMD of the root node of the tree T. The notion of Tree_MD is defined as follows :

Tree_MD : The Tree_MD of a node N is recursively defined based on the Node_MD of the node N and Tree_MDs of children of the node N. The Tree_MD is defined precisely using nMD and tMD as follows :

$$\text{tMD}(N) = \text{string}(\text{MD4}(\text{nMD}(N) \bigoplus_{i=1}^{n} \text{tMD}(\text{Child\_}i(N))))$$

where n is the number of child nodes, and Child_i (N) returns ith child node of node N.

From now on, nMD (also, tMD) are used to represent both the field name and Node_MD (also, Tree_MD). (Figure 4) shows how nMD and tMD for nodes are computed in a sample X-tree, where node N_i has only one child node N_k, also a leaf node. Note that tMD(N_i) is defined on both nMD(N_i) and tMD(N_k) which, recursively, is defined on nMD(N_k). Due to the recursive definition, the tMD of an X-tree contains not only the structure of the tree, but also contents of all the nodes in a tree. Therefore, if the roots of two X-trees have the same tMD, these trees are assumed to be *identical*.



(Figure 4) An example of Node_MD and Tree_MD

2.3 Edit Operations

When two X-trees are not identical, change detection algorithms should find the difference between these trees. The difference between X-trees can be described with a set of edit operations. In this paper, we consider four basic edit operations such as *INS*, *DEL*, *UPD*, and *MOV*.

For an X-tree T, these edit operations are defined as follows :

- DEL(nID(N)) : delete a node N from X-tree T. However, we assume that the root of T cannot be deleted.
- INS(l, v, nID(N), i) : create a new node that has the value l for the Label field and the value v for the Value field, then insert the node to X-tree T as the ith child of node N.
- UPD(nID(N), v) : update the Value field of a node N with the value v.
- MOV(nID(N), nID(M), j) : move the subtree rooted at a node N from T, and make the subtree being the jth child of M.

Note that nIDs are used in the definitions, instead of node pointers. These are logical definitions, so that we can use them in any change detection system. The set of edit operations in our model is similar to the one in BULD Diff algorithm [8]. In addition to operations above, we also use *NOP* in X-tree Diff, which is a dummy operation for nodes with no change occurred.

Changes in two XML documents can be viewed as the difference between the old version and the new version. From the functional view, applying an edit operation to an X-tree

7) If the names and values of attributes are stored in the Value field, Node_MD may be much more useful.

8) BULD Diff algorithm supports the same set of operations. However, their insert and delete operations are applied to subtrees, instead of nodes.

will produce a new X-tree. Also, applying a sequence of edit operations to the old version of an X-tree means transforming the old version, through a sequence of temporary versions, to the new version. From this viewpoint, changes in XML documents can be represented by a list of edit operations.

3. X-Tree Diff : Change Detection Algorithm

In this section, we introduce X-tree Diff, the change detection algorithm, which matches nodes from two X-trees and finally compute the difference. In Section 3.1, we describe the outline of the X-tree Diff algorithm. In Section 3.2, we discuss about the preliminary step before X-tree Diff begins, i.e., how to initialize data structures used in X-tree Diff as well as all the fields of X-tree nodes. At last, we present detailed explanation on X-tree Diff in Section 3.3.

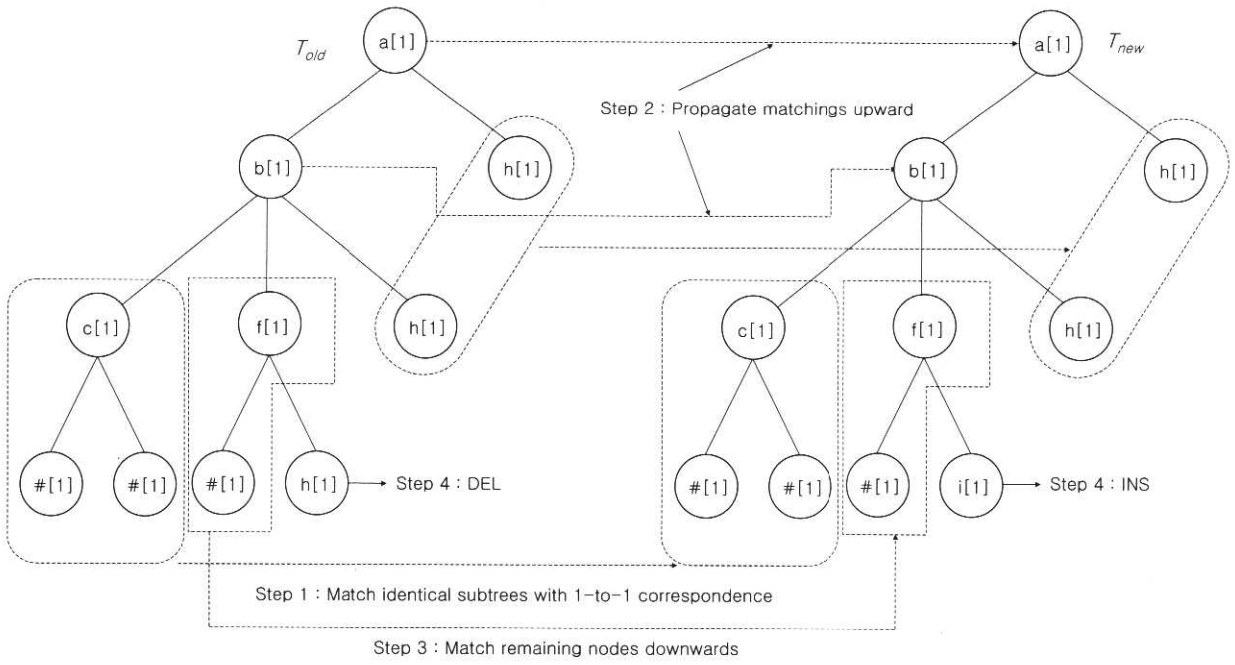
3.1 Overview

To begin with, we explain about the notion of matching. Matching two nodes means that these nodes are made to correspond to each other. It also implies that there is a proper edit operation involved with these nodes, even though sometimes it is not mentioned explicitly. The edit operation may be NOP, UPD, or MOV, depending on values of the nMD, Label, and Value fields of nodes involved. We present some definitions about the notion of matching. If an edit operation e needs mentioning for a pair (N_i, N_j) of nodes being matched, it is said that that node N_i and node N_j are matched using edit operation e. For the nodes N_i and N_j, and edit operation e, a matching is represented by a tuple (N_i, N_j) or a triple (N_i, N_j, e), depending on the context. The triple (N_i, N_j, e) is often written as "the tuple (N_i, N_j) with e". Node N_i is called the matching node of node N_j, and vice versa. For example, if A is matched with B using NOP, it may be written as '(A, B, NOP)', or '(A, B) with NOP'. Also, A is called a matching node of B.

The matching between two nodes can be often interpreted as the matching between two trees, due to the existence of tMD fields in X-trees. This extension is very useful to match identical subtrees, since we can match two identical subtrees by simply matching their roots using NOP.

Briefly speaking, X-tree Diff algorithm proceeds in the following four steps. At first, it finds matchings with NOP (i.e., pairs of matched identical subtrees) among all the pairs of identical subtrees from T_{old} and T_{new}. Next, it propagates these matchings upward to the root. In Step 3, downwards from the roots, in depth-first order, it attempts to match nodes from unmatched nodes in T_{old} and T_{new}. Finally, it determines INS or DEL operations for nodes remaining unmatched up to this point, and generates a report to users.

These four steps are illustrated in (Figure 5). As shown



(Figure 5) Illustration of X-tree Diff algorithm

before, labels of text nodes begin with '#', while element nodes do not. In the example, it is assumed that the Value fields of the leftmost two text nodes remain unchanged, and that of the third text node has been changed. We also assume that the Label fields of all the other element nodes remain unchanged except ones to be deleted or inserted. Before explaining the algorithm in detail, we will sketch how the algorithm works based on the example in (Figure 5).

In Step 1, the subtree rooted at '.a[1].b[1].c[1]' of T_{old} is matched with the subtree rooted at '.a[1].b[1].c[1]' of T_{new}, since they are identical subtrees. At Step 2, the matching is propagated upwards to nodes of '.a[1].b[1]' of T_{old} and T_{new}, and again, to nodes of '.a[1]'. Note that these nodes could not be matched in Step 1, since some of their descendents have been changed. In Step 3, other nodes are matched in dept-first order, except two nodes to be deleted and inserted. Detailed explanation will be provided later. Finally, the node '.a[1].b[1].f[1].h[1]' of T_{old} is determined to be deleted, while the node '.a[1].b[1].f[1].i[1]' of T_{new} to be inserted.

In the algorithm, information about matchings is stored in the nPtr and Op fields of X-tree nodes. Specifically, in the nPtr and Op fields of an X-tree node, the pointer of its matching node and the name of edit operation involved in the matching are stored, respectively. For example, in the nPtr and Op fields of the node '.a[1].b[1].f[1].#[1]' of T_{old}, the pointer of '.a[1].b[1].f[1].#[1]' of T_{new} and the string "UPD" are stored, respectively. Note that the name of edit operations is stored in the Op fields.

3.2 Building X-trees and Hash Tables

In this section, we explain about the preprocessing step for X-tree Diff. During the buildup of X-trees T_{old} and T_{new} from XML documents, we generate two hash tables used in later steps. We present detailed explanation.

Step 0 (Build up X-Trees) : In this step, we convert XML documents into X-trees T_{old} and T_{new}, and generate hash tables used in X-tree Diff. During this process, all the Label, Index, and Value fields of X-tree nodes are properly initialized. In addition, for each one of all the nodes from T_{old} and T_{new}, its nPtr and Op fields are set to NULL, and its nMD and tMD, after being computed, are stored in the nMD and tMD fields, respectively.

During the preprocessing process, two hash tables O_Htable and N_Htable are generated. Entries for both N_Htable and O_Htable are of tuples consisting of the tMD and pointer of a X-tree node, with tMD as the key. All the nodes with unique tMD in T_{new} are registered to N_Htable, while all the nodes with non-unique tMD in T_{old} are registered to O_Htable. These hash tables are used to match nodes with one-to-one correspondence from T_{old} to T_{new} in Step 1.

3.3 X-tree Diff Algorithm

As described before, X-tree Diff consists of four steps.

Step 1 (Match identical subtrees with 1-to-1 correspondence) : In this step, we find pairs of identical subtrees among all subtrees from T_{old} and T_{new}, and match them using NOP. Let a pair of matched identical subtrees be (ST_{old}, ST_{new}). When ST_{old} and ST_{new} are matched using NOP, all the Op fields of nodes in ST_{old} and ST_{new} are set to NOP, and the nPtr field of each node in ST_{old} is set to the pointer of the corresponding node in ST_{new}, and vice versa.

The algorithm in (Figure 6) describes this process, in detail. In the algorithm, N and M represent the roots of subtrees in T_{old} and T_{new}, respectively. At the end, the algorithm computes the list of matchings, called *M_List*, which is the input data for Step 2. Note that, after finding a pair of matched nodes(a matching), we don't visit their subtrees for matching. It is obvious from the characteristics of tMD and depth-first order traversal.

The first 'If' condition tests whether the tMD of node N is unique in T_{old}, and the second one tests whether there exists some entry with the same tMD in N_Htable. These conditions guarantees that tMD(N) and tMD(M) are unique in T_{old} and T_{new}, respectively, and tMD(N) = tMD(M). Note that all the nodes with unique tMD in T_{new} are registered to N_Htable, while all the nodes with non-unique tMD in T_{old} are registered to O_Htable. Therefore, the pairs of nodes (matchings) found in this process have one-to-one correspondence. For a node N in T_{old}, if we attempt to match the node N with a node in T_{new}, which has one-to-many (or many-to-one) correspondence, then we might end up with wrong matchings. The motivation behind this is : unless matching between nodes is obvious, delay it until ambiguity in the matching disappears or until last moment for the decision. This is called *the Rule of Delaying Ambiguous Matchings*. It is why subtrees rooted at '.a[1].b[1].c[1]' in T_{old} and T_{new} are matched in Step 1, and nodes whose lLabel value is 'h[1]' remain unmatched after Step 1 in (Figure 5).

```
/* Visit each node of T_old in depth-first order */
For each node N in T_old {
    If any entry of O_Htable does NOT have the same tMD value
            that the node N has then {
        If some entry of N_Htable has the same tMD value that
                the node N has then {
            Retrieve the node M from N_Htable ;
            Match the nodes N and M using NOP ;
            /* also match corresponding their descendents */
            Add the pair (N, M) of nodes to M_List ;
            Stop visiting all the subtrees of the node N, then go
                on to next node in T_old ;
        }
        Else
            Go on to next node in T_old ;
    }
    Else
        Go on to the next node in T_old ;
} // For
```

(Figure 6) Matching identical subtrees with one-to-one correspondence

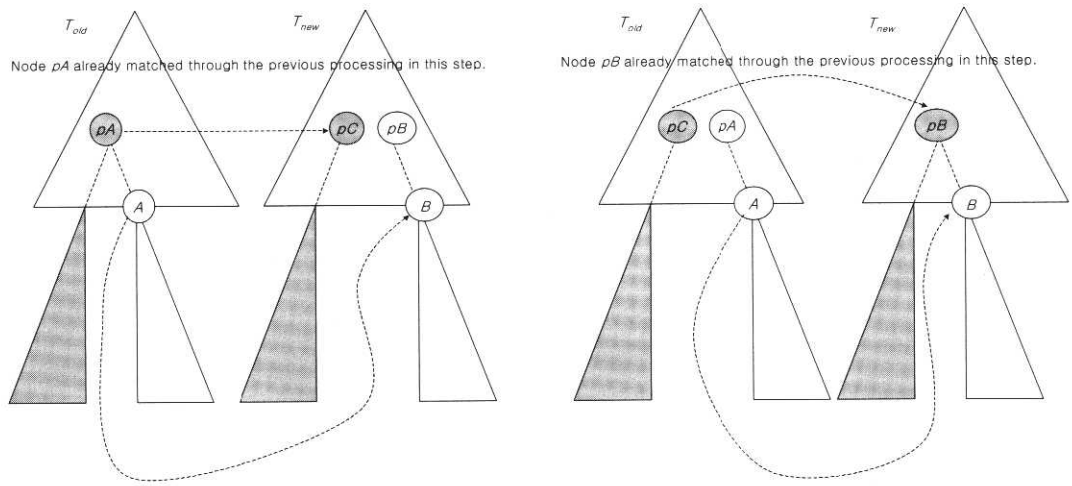
Step 2 (Propagate matchings upward) : In this step, we propagate the matchings (i.e., pairs of matched subtrees) found in Step 1 upward to the roots. The algorithm of Step 2 is provided in (Figure 7).

For each matching (A, B) found in Step 1, let pA and pB denote parents of A and B, respectively. Now, we decide whether pA can be matched with pB. First, consider the case that none of pA and pB have been matched. When Label (pA) is equal to Label (pB), match pA and pB using NOP, then continue to propagate the matchings toward their parents. However, if Label (pA) is not equal to Label(pB), match A and B using MOV, after revoking the previous matching $(A, B,$ NOP) for A and B, leaving pA and pB unmatched. Note that 'rematch A and B' in (Figure 7) revokes the previous matchings made for A and B before 'match' them again. Then go on to the next pair from M_List.

```
/* Propagate each matching from M_List to its parents */
For matching (A, B) in M_List from Step 1 {
    pA = Parent (A) ; pB = Parent (B)
    While TRUE {
        /* None of parents have been matched. */
        If nPtr (pA) == NULL AND nPtr (pB) == NULL then {
            If Label (pA) == Label (pB) then
                Match pA and pB using NOP ; pA = Parent (pA) ;
                    pB = Parent (pB) ;
            Else
                /* make the subtree rooted at A to be a child of pB. */
                Rematch A and B using MOV ; Break ;
        }
        Else {
            /* Case A : pA has been matched but not with pB. */
            If nPtr(pA) != NULL AND nPtr(pA) != pB then
                Rematch A and B using MOV ; Break ;
            /* Case B : pB has been matched but not with pA. */
            Else if nPtr(pB) != NULL AND nPtr(pB) != pA then
                Rematch A and B using MOV ; Break ;
            /* Case C : pA has been matched with pB. */
            Else
                Break ;
        }
    } // While
} // For
```

(Figure 7) Propagate matchings upward

Now, consider the case that at least one of pA and pB have been matched, which is corresponding to the outermost 'Else' part in the algorithm. Suppose that pA has been matched with some node which is not pB, which is *Case A* in (Figure 7). This situation occurs, when pA has been matched



(a) *Case A* in the algorithm (b) *Case B* in the algorithm

(Figure 8) Resolve the conflict of matchings propagated from different subtrees

by propagating a matching from one of its subtrees except the one rooted at A, as shown in (Figure 8) (a). In this case, we preserve the matching (pA, pC) and matchings represented by the shaded subtree, and we 'Rematch A and B' using MOV. *Case B* in (Figure 7) represents the situation that pB has been matched with some node which is not pA. The similar conflict resolution is applied to this case, as shown in (Figure 8) (b).

The final 'Else' part represents the case where pA has been already matched with pB by matching propagation from other subtrees. It implies the condition that nPtr (pA) = pB (also, nPtr (pB) = pA). It follows from negations of all the conditions in 'If' parts in (Figure 8) : NOT (nPtr (pA) == NULL AND nPtr (pB) == NULL) AND NOT (nPtr (pA) != NULL AND nPtr (pA) != pB) AND NOT (nPtr (pB) != NULL AND nPtr (pB) != pA). In this case, we don't need to propagate matching (A, B, NOP) upwards, since matching (pA, pB, NOP) has been generated from the propagation from other matchings. So, just go on to the next pair from M_List. Note that nodes of '.a[1].b[1]', again nodes of '.a[1]' in T_{old} and T_{new} in (Figure 5) are matched through this step.

Step 3 (Match remaining nodes downwards) : In this step, downwards from the roots, we attempt to match nodes remaining unmatched until Step 3 begins.

While visiting nodes in T_{old} in depth-first order, we repeat the following :

① Find a matched node which has unmatched children. Let A be the node in T_{old} and B be the matching node of A (i.e., nPtr (A)). For A and B, let cA [1..s] and cB [1..t] denote the list of unmatched child nodes of A, and the list of unmatched child nodes of B, respectively, where s is the number of unmatched child nodes of A and t is also similarly defined.

② For A, B, cA [1..s], and cB [1..t], perform the algorithm in (Figure 9).

The 'then' part of the first 'If' clause in (Figure 9) implies that we now process nodes that have been excluded for matching process of Step 1. Note that they have not been registered to N_Htable in Step 0, due to *the Rule of Delaying Ambiguous Matchings*. Since matching space for these nodes becomes much smaller than before, the quality of matchings for these nodes is improved. The nodes, in (Figure 5), whose ILabel value is 'h[1]', are matched in this step.

The 'Else if' part represents that, among unmatched child nodes of A, we choose a node whose ILable value is equal to ILabel value of cB [j]. It implies the following : for each node in cB[1..t], the Label value is used as the first criterion to find a matching in cA [1..s], and if there are nodes with the same Label value in cA [1..s], then the relative position among cA [1..s] is used as the second criterion. The nodes '.a [1].b [1].f [1].# [1]' in (Figure 5), are matched using UPD at this stage. Note that we assume its Value field has been changed.

Two hash tables are used in this process. When a node A is found in T_{old}, all the unmatched child nodes cA[1..s] are registered to both hash tables, where one hash table uses the tMD values of nodes as the key, and the other the ILabel values as the key. These hash tables are used to find matchings for each unmatched child node cB [j] of node B in T_{new}.

```
/* For each cB[j] */
For j in [1..t] {
    If there is a cA[i] whose tMD value is equal to tMD value
            of cB[j] then
        Match subtree rooted at node cA[i] and subtree rooted
            at node cB[j] using NOP ;
    Else If there is a cA[i] whose ILabel value is equal to ILabel
            value of cB[j] then {
        If Value (cA[i]) is equal to Value (cB[j]) then
            Match cA[i] and cB[j] using NOP ;
        Else
            Match cA[i] and cB[j] using UPD ;
    }
} // For
```

(Figure 9) Match remaining nodes downwards

Step 4 (Determine nodes for addition and deletion) : In this step, we assume that all nodes that can be matched have been matched through Step 1 ~ Step 3. Therefore nodes that remain unmatched in T_{old} are marked for deletion, while nodes unmatched in T_{new} for insertion. In order to do these tasks, we traverse T_{old} to find unmatched nodes and set Op fields to DEL, while we traverse T_{new} to find unmatched nodes and set Op fields to INS.

Next, we generate reports to users. Since generating reports is straightforward, we are not going to explain report generation algorithm here.

4. Algorithm Analysis and Experiments

In this section, we investigate the time complexity of our algorithm and present the result of experiments performed on X-tree Diff. Let us analyze the complexity of our algo-

rithm first. $|T|$ denotes the number of nodes in tree T.

In Step 0, we construct X-trees T_{old} and T_{new} and two hash tables O_Htable and N_Htable. Since the time cost of constructing X-trees is bounded by $O(|T_{old}|+|T_{new}|)$ and also that of the hash tables is the same, the overall time cost of Step 0 is bounded by $O(|T_{old}|+|T_{new}|)$.

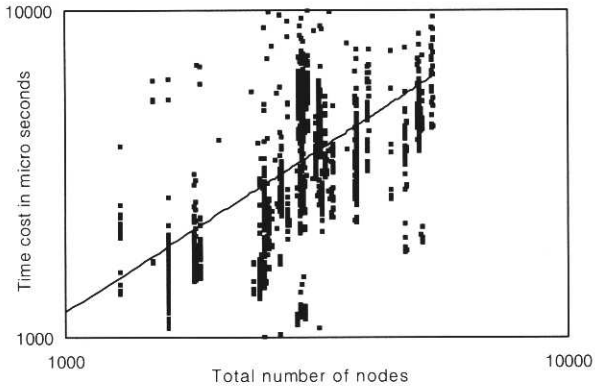
In Step 1, during the depth-first ordered traversal of T_{old}, for each node visited, we look up O_Htable and N_Htable to see whether a matching is found. If not found, we go on to next nodes. But if found, for all the nodes belonging to matched subtrees, we do not look up hash tables any more, but update their nPtr and Op fields. In both cases, we need to visit every node in T_{old}. Note that the time cost for the tree traversal is bounded by $O(|T_{old}|)$. Since both the cost of 'look up' operation to hash tables and that of updating the nPtr and Op fields of a node is $O(1)$, the overall cost of Step 1 is bounded by $O(|T_{old}|)$.

In Step 2, we propagate the matchings found in Step 1 upward to roots. The worst case occurs when all the internal nodes are visited for this process. For each node visited, if a matching is found, the nPtr and Op fields are updated, which costs $O(1)$ as seen before. Therefore, the time cost in Step 2 is bounded by $O(|T_{old}|)$.

In Step 3, we attempt to visit nodes of T_{old} in depth-first order and match nodes remaining unmatched. The worst case for this step is that all the nodes, except the root, in T_{old} remain unmatched at the beginning, and all these nodes become matched at the end [9]. This case occurs when Value fields of all the nodes in T_{old} have been updated except the root node. In order to process this case, we need to traverse T_{old} and T_{new}. As known before, the time cost for tree traversals is bounded by $O(|T_{old}|+|T_{new}|)$. In addition, all nodes except the root in T_{old} are registered to hash tables in the worst case. The time cost for this task is also $O(|T_{old}|)$. Also, the cost of hashing and updating the nPtr and Op fields of matched nodes is $O(1)$. Therefore, the overall time cost becomes bounded by $O(|T_{old}|+|T_{new}|)$.

In Step 4, we traverse T_{old} and T_{new} to find unmatched nodes and set Op fields properly. The time cost of these is $O(|T_{old}|+|T_{new}|)$. The cost of report generation is also to $O(|T_{old}|+|T_{new}|)$, since report generation requires traversal of T_{old} and T_{new}.

In conclusion, the time cost of X-tree Diff, even in worst case, is $O(|T_{old}|+|T_{new}|)$. That is, time complexity is proportional to the number of all the nodes in the two X-trees. It is better than the time cost of the BULD Diff algorithm.



(Figure 10) Result of experiments

Now, we mention about experiments carried out on X-tree Diff. We implemented X-tree Diff algorithm in MS Visual C++, using Xerces C++ XML parser [10], which is the same parser used by the BULD Diff [16] and X-Diff+ [17]. We experimented X-tree Diff on sixties web pages of twenties web sites, which are of newspapers and TV/Radio stations. In (Figure 10), results of experiments on about 11,000 old and new versions from these web pages are reported.

For each one of points in (Figure 10), the x-coordinate represents the number of nodes in one of experimented HTML documents, while the y-coordinate the execution time to detect changes between the old and new version of the same document. Note that the trend line in (Figure 10), which is acquired by linear regression analysis, shows that the time cost of our algorithm is proportionate to the number of nodes in documents. These experiments were carried out on a Pentium III 1GHz PC with 384MB memory in MS windows 2000.

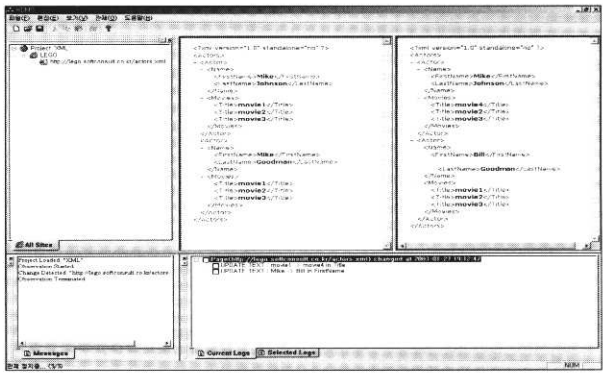
Currently, X-tree Diff algorithm is being used in a commercial hacking detection system, called the WIDS, which is to find changes occurred in registered websites, and report suspicious changes to users. In (Figure 11), we show a screen of WIDS where, for the old and new version of an XML document, changes are detected and reported to users.

Since August in 2002, over 7000 web pages from about 700 web sites have been registered to WIDS, and these web pages, currently, are being monitored every 5 to 20 minutes, depending on user requirements. According to our experi-

9) We assume that roots of T_{old} and T_{new} are always matched.

10) Available at http://xml.apache.org/xerces-c.

ence, X-tree Diff is fast enough, and the output quality is quite satisfactory.



(Figure 11) A snapshot of WIDS

5. Conclusion and Future work

In this paper, we propose a change detection algorithm, X-tree Diff, which is fast, reliable, simple and clear. It is shown that X-tree Diff runs fast in time cost in worst case, which is even faster than BULD Diff algorithm [16]. Because of using MD4 [20] as hash function and *the Rule of Delaying Ambiguous Matchings*, matching process in X-tree Diff is more reliable than any other algorithms using hashing [15-17], so that the possibility of wrong matchings may be reduced.

In this work, we experimented X-tree Diff on real data instead of synthetic documents generated for experimentation. According to experiments of applying our algorithm to over 11,000 old and new versions from sixties web pages of twenties web sites, the performance is reported as satisfactory. We are currently working on the comparison of X-tree Diff with existing algorithms in terms of output quality, based on both real and synthetic data. The result will be published in a forthcoming paper.

The structure of X-tree Diff is so simple and clear that we can extend it in various ways. Right now, we are interested in tuning X-tree Diff by extending it according to the requirements of applications.

References

[1] A. Haake, "CoVer : A Contextual Version Server for Hypertext Applications," *In Proc. of 4th ACM Conf.*, Hypertext, Milan. Italy, pp.43-52, Nov., 1992.

[2] K. Osterbye, "Structural and Cognitive Problems in Providing Version Control for Hypertext," *In Proc. of 4th ACM Conf.*, Hypertext, Milan. Italy, pp33-42, Nov., 1992.

[3] W. Labio and H. G. Molina, "Efficient snapshot differential algorithms for data warehousing," *In Proc. of 20th Conf. VLDB*, Bombay. India, pp.63-74, Sep., 1996.

[4] J. Widom and S. Ceri, Active Database System : Triggers and Rules for Advanced Database Processing, Morgan Kaufmann, 1996.

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

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

[7] S. Chawathe, A. Rajaraman, H. G. Molina and J. Widom, "Change Detection in Hierarchically Structured Information," *In Proc. of ACM SIGMOD Int'l Conf. on Management of Data*, Montreal, June, 1996.

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

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

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

[11] J. T. Wang and K. Zhang, "A System for Approximate Tree Matching," *IEEE TKDE*, 6(4), pp.559-571, August, 1994.

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

[13] S. Chawathe, "Comparing Hierarchical Data in External Memory," *In Proc. of the 25th VLDB Conf.*, pp.90-101, 1999.

[14] S. J. Lim and Y. K. Ng, "An Automated Change-Detection Algorithm for HTML Documents Based on Semantic Hierarchies," *The 17th ICDE*, pp.303-312, 2001.

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

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

[17] Y. Wang, D. J. DeWitt, J. Y. Cai, "X-Diff : An Effective Change Detection Algorithm for XML Documents," *To appear in the 19th ICDE*, 2003.

[18] 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.

[19] H. Maruyama, K. Tamura and N. Uramoto, "Digest values for DOM(DOMHash) Proposal," *IBM Tokyo Research Laboratory*, 1998.

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

[21] N. Doraswamy and D. Harkins, IPSec : The New Security Standard for the Internet, Intranets, and Virtual Private Networks, Prentice Hall PTR, 1999.

[22] Document Object Model (DOM) http://www.w3.org/DOM/.

[23] R. Wagner and M. Fischer, "The string-to-string correction problem," *Journal of the ACM*, 21, pp.168-173, 1974.

이 석 균

e-mail : sklee@dankook.ac.kr

1982년 서울대학교 경제학과(학사)

1990년 Computer Science, University of Iowa(석사)

1993년 Computer Science, University of Iowa(박사)

1992년 국제학술대회 ICDE(Int. Conf. On Data Engineering)에서 최우수 논문상 수상

1993년~1997년 세종대학교 정보처리학과 교수

1997년~현재 단국대학교 컴퓨터과학과 부교수

관심분야 : 데이터베이스에서 불완전 정보관리, 데이터베이스 시각 질의어 설계, 다중처리기에서의 실시간 스케줄링, XML, 웹 문서에서의 변화 탐지, 버전닝(versioning) 등

김 동 아

e-mail : dakim70@dankook.ac.kr

1993년 서울교육대학(학사)

1999년 단국대학교 교육대학원 전산교육과(석사)

2002년 단국대학교 대학원 컴퓨터과학과(박사수료)

2000년~2002년 (주)소프트웨어컨설팅그룹 선임연구원

2003년~현재 화랑초등학교 교사

관심분야 : 데이터베이스 모델링, 전산교육, XML, 웹 문서에서의 변화 탐지, 버전닝(versioning) 등