X-tree Diff+: Efficient Change Detection Algorithm in XML Documents[*]

Suk Kyoon Lee[°], Dong Ah Kim

Division of Information and Computer Science, Dankook University
San 8, Hannam-dong, Youngsan-gu, Seoul, 140-714, Korea
{sklee, dakim70}@dankook.ac.kr

Abstract. As web documents proliferate fast, the need fo real-time computation of change (edit script) between web documents increases. Though fast heuristic algorithms have been proposed recently, the qualities of edit scripts produced by them are not satisfactory. In this paper, we propose X-tree Diff+ which produces better quality of edit scripts by introducing a tuning step based on the notion of consistency of matching. We also add copy operation to provide users more convenience. Tuning and copy operation increase matching ratio drastically. X-tree Diff+ produces better quality of edit scripts and runs fast equivalent to the time complexity of fastest heuristic algorithms.

1 Introduction

Computing change (edit script) between documents draws much attention from Computer Science research community. It is because version management becomes important, as Internet is crowded with excessive information. Version management is based on the algorithm of computing change between web documents [10]. It is also used in the real-time detection of hackers' defacement attacks [14]. In this paper, we present an efficient change detection algorithm X-tree Diff+ for tree-structured documents such as HTML, XML documents. Difference (change) between documents can be viewed as a sequence of edit operations (called edit script). Researches on change detection for tree-structured data such as XML/HTML documents have been carried out since late 1970s [1,2,3,4,5,6,7,8,9,10,11,12]. Early works focused on computing the minimal cost edit script for tree-structured data [2,3,4]. The general problem on change detection for tree-structured data is known as *NP*-hard [6]. So, recently, heuristic algorithms have been proposed to meet the demand. XyDiff [10] is designed for XML data warehouse and versioning. This algorithm uses hashing to represent the contents and structure of subtrees, and the notion of weigh in matching process. It achieves $O(n \log n)$ complexity in execution time. X-Diff+ [11] uses hashing to represent the contents and structure of subtrees. X-Diff+ computes the edit distance between two documents, and produce reasonable good edit scripts. But the time cost of X-Diff+ is not acceptable for real-time application. X-tree Diff [14] is

[*] The present research was conducted by the research fund of Dankook University in 2005.
[°] The corresponding author.

designed to detect hackers' defacement attacks to web sites. X-tree Diff is simple and runs fast in $O(n)$. XyDiff and X-tree Diff are quite efficient in execution time. However, the quality of edit scripts produced by them is not good enough.

In this paper, we propose X-tree Diff+ based on X-tree Diff. It produces better quality of edit scripts and run in $O(n)$. We introduce the notion of consistency of matching to tune ineffective matches. Also, we add Copy operation as a basic edit operation. With these, X-tree Diff+ increases matching ratio and produce better edit scripts. We present the change representation model in Section 2, and X-tree Diff+ algorithm in Section 3. We analyze the time complexity of X-tree Diff+ in Section 4.

2 The Model for Representing Changes in XML Documents

2.1 X-tree

X-tree is a labeled ordered tree. The logical structure of a node of X-tree consists of nine fields shown in Figure 1. *Label*, *Type* and *Value* fields are used to 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*, *iMD*, *nPtr* and *Op* fields for matching process in X-tree Diff+. iMD field is added in the X-tree in X-tree Diff+. A node in an XML document is represented by an X-tree node. For each XML text node, the value 0, the string "#TEXT" and the text content of the node are stored, respectively, in the Type field, the Label field and Value field of the corresponding X-tree node. For each XML element node, the value 1 and the name of the node is stored in the Type field, the Label field of the corresponding node. For an XML element node with a list of attributes, the list of each pair of attribute name and attribute value is stored as a text in the Value field. For 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.

Label	Type	Value	Index	nMD	tMD	iMD	nPtr	Op

Fig. 1. Logical structure of a node of X-tree.

The iMD field represents ID attribute value for a node in XML documents. ID attributes, if used with Label, can uniquely identify each node in an XML document. The iMD, nMD and tMD fields are used for efficient comparison between two X-trees τ and τ', converted from two XML documents. The nMD field of an X-tree node N contains the hash value for information of node N and the tMD field of node N the hash value for information of the node N and its descendents. For the notational simplicity, a field name of X-tree node is used as a function taking X-tree node as an input and returning the corresponding field value. iMD(N), nMD(N) and tMD(N) are defined as follows:

$$\textbf{iMD(}\boldsymbol{N}\textbf{)} = \text{hash}(\text{Label}(\tau_i) \oplus \text{`ID attribute value'})$$

$$\textbf{nMD(}\boldsymbol{N}\textbf{)} = \text{hash}(\text{Label}(N) \oplus \text{Value}(N)) \quad \textbf{tMD(}\boldsymbol{N}\textbf{)} = \text{hash}(\text{nMD}(N) \bigoplus_{x=1}^{n} \text{tMD}(C_x(N)))$$

where $\oplus$ is a string concatenating operator, n is the number of child nodes of node N, and $C_x(N)$ returns xth child node of node N. Note that tMD(N) reflects the structure and contents of a subtree rooted at node N. Therefore, if the roots of two X-trees have the same tMD, these trees are assumed to be *identical*.

2.2 Edit Operations and the notion of matching

Applying an edit script to the old version of an X-tree produces the new version from the old version. In this paper, we consider five edit operations such as *INS*, *DEL*, *UPD*, *MOV*, and *COPY*. These edit operations are defined as follows:

- DEL(N): delete a node N from X-tree T.
- INS(l, v, N, i): create a new node with 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(N, v'): update the Value field of a node N with the value v'.
- MOV(N, M, j): remove the subtree rooted at a node N from T, and make the subtree being the jth child of M.
- COPY(N, M, j): copy the subtree rooted at a node N to the jth child of M.

Copy operation is added in X-tree Diff+. Copy operation is useful when same subtrees are scattered in duplicate around an X-tree. It allows users to use copy operations instead of a sequence of insert operations. In addition, we also use *NOP* in matching process, a dummy operation for nodes with no change occurred. Matching two nodes means that these nodes are made to correspond to each other with some edit operation involved in the match. For a pair of matched nodes N_i, N_j with operation e, it is said that that node N_i are matched with node N_j using edit operation e. Also, it is represented by a triple (N_i, N_j, e) or $N_i \xrightarrow{e} N_j$. When e is NOP, e is often omitted. Let's define several functions. For a node N, $M(N)$ and $P(N)$ return the matching node and parent node of node N, respectively. $ST(N)$ returns a subtree rooted at node N. If tMD(N_i) = tMD(N_j), then matching between the subtree rooted at N_i and the subtree rooted at N_j may be represented as $ST(N_i) \rightarrow ST(N_j)$.

3 X-Tree Diff+: Change Detection Algorithm

In this section, we explains X-tree Diff+. In Figure 2, X-tree τ is an old version, and X-tree τ' a new version. The number beside a node of these X-trees represents the order of visiting the node in post-order traversal. These numbers are used to identify each node in the tree. For example, τ_1 represent the leftmost node in τ.

3.1 Preprocessing

Step 0 (Build up X-Trees): we convert XML documents into X-trees τ and τ', and generate hash tables. During this process, all the fields of X-tree nodes are properly initialized. In addition, for each X-tree node, nMD, tMD and iMD values are computed and stored in the nMD, tMD and iMD fields, respectively.

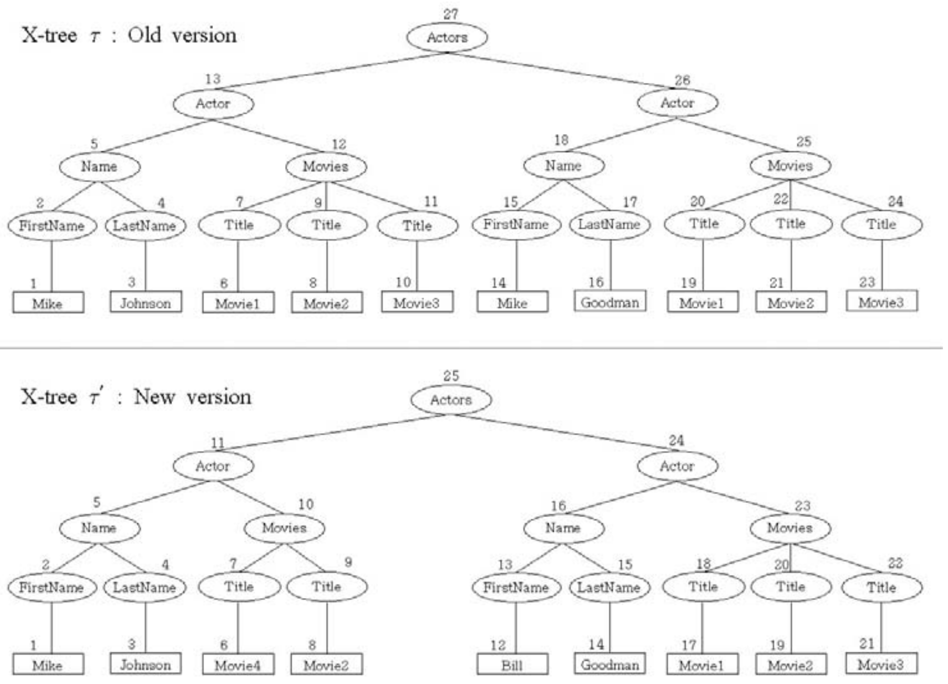


Fig. 2. Examples of X-tree

Three hash tables O_Htable, N_Htable, and N_IDHtable are generated. Entries for first two hash tables are of tuples consisting of the tMD and pointer of an X-tree node, with tMD as the key. All the nodes with unique tMD value in τ' are registered to N_Htable, while all the nodes with non-unique tMD in τ are registered to O_Htable. N_IDHtable is an hash table for nodes with ID attributes in X-tree τ'. The entry consists of iMD(as a key) of a node and a pointer to the node.

3.2 X-tree Diff+ Algorithm

Step 1 (Match identical subtrees with 1-to-1 correspondence and match nodes with ID attributes): First we repetitively find a pair of identical subtrees one from τ and another from τ', and match them using NOP. The algorithm in Figure 3 describes this process. *N* and *M* represent the roots of subtrees in τ and τ', respectively. At the end, the algorithm computes the list of matches, called *M_List*, which is the input data for Step 2. Note that, after finding a match, we don't visit their subtrees for matching, due to the characteristics of tMD. For the details, refer to [14,17]. The matches found in this process have one-to-one correspondence.

After having finished the previous sub-step, we try to match nodes with a same iMD values. While traversing X-tree τ in breadth-first order, if unmatched node with ID attribute is found, then look up an entry in N_IDHtable with the same iMD value. If the lookup succeeds, then match them with NOP.

```
For each node N in τ { /* Visit each node of τ in breadth-first order */
  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;
          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 τ ;   }
      Else Go on to next node in τ ;   }
  Else Go on to the next node in τ ;
} // For
```

Fig. 3. Matching identical subtrees with one-to-one correspondence

In Figure 2, since both ST(τ_5) and ST(τ_{17}) are unique in τ, they can be matched with ST(τ'_5) and ST(τ'_{15}) of X-tree τ', respectively. So matches such as ST(τ_5)→ST(τ'_5) and ST(τ_{17})→ST(τ'_{15}) are produced. Also, one of ST(τ_{12}) and ST(τ_{25}) can be matched with ST(τ'_{23}). But since ST(τ_{12}) and ST(τ_{25}) are not unique in the X-tree τ, this match cannot be determined at this step.

```
/* Propagate each matching from M_List to its parents */
For matching (A, B) in M_List from Step 1 and 2 {
  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 Break;        }
      Else Break;
  } // While
} // For
```

Fig. 4. Propagate matching upward

Step 2 (Propagate matching upward): In this step, we propagate matching from matches found in step 1 upward to the roots. For each matching *A*→*B* determined in step 1, we need to decide whether the parent (*P*(*A*)) of *A* can be matched with the parent (*P*(*B*)) of *B* based on their labels. The algorithm is shown in Figure 4.

In the example shown in Figure 2, we propagate the matching $\tau_5 \rightarrow \tau'_5$ found in step 1 upward. Since their parent nodes have the same label, a matching $\tau_{13} \rightarrow \tau'_{11}$ is made in this step. Again, since parent nodes of nodes in $\tau_{13} \rightarrow \tau'_{11}$ have the same label, then a matching $\tau_{27} \rightarrow \tau'_{25}$ is determined. In similar way, from the matching $\tau_{17} \rightarrow \tau'_{15}$ from Step 2, $\tau_{18} \rightarrow \tau'_{16}$ and $\tau_{26} \rightarrow \tau'_{24}$ are produced.

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 τ in depth-first order, we repeat the following:

(1) Find a matched node *A* in τ which has unmatched children. Let *B* be *M*(*A*), the matching node of *A*. For *A* and *B*, let *cA*[1..*s*] denote the list of unmatched child nodes of *A*, and *cB*[1..*t*] the list of unmatched child nodes of *B*, 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 5.

In the implementation of Step 3, two hash tables are used. When a node *A* is found in τ, 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 Label and Index values as the key. These hash tables are used to find matching for each unmatched child node *cB*[*j*] of node *B* in τ'.

```
For j in [1..t] {   /* For each cB[j] */
  If there is a cA[i] where tMD(cA[i]) = tMD(cB[j]) then
      Match ST(cA[i]) with ST(cB[j]) using NOP;
  Else If there is a cA[i] where Label(cA[i]) = Label(cB[j]) and Index(cA[i]) = Index(cB[j]) then
       Match cA[i] with cB[j] using NOP;
  }     } // For
```

Fig. 5. Match remaining nodes downwards

In the example in Figure 2, while visiting nodes in τ in the depth-first order, we first find an matched node τ_{13} where τ'_{11} = *M*(τ_{13}). Nodes τ_{13} and τ'_{11} have unmatched child τ_{12} and τ'_{10}, respectively. Because these children have the same label, then a mach $\tau_{12} \rightarrow \tau'_{10}$ is made. Similarly, node τ_{12} has unmatched children(τ_7 , τ_9 , τ_{11}) while node τ'_{10} have unmatched children(τ'_7 , τ'_9). First of all, because node τ_7 and node τ'_7 have the same label and index values, then a match $\tau_7 \rightarrow \tau'_7$ is generated. Also, from this match ($\tau_7 \rightarrow \tau'_7$), $\tau_6 \rightarrow \tau'_6$ is produced, because all text nodes have the same label (TEXT). Second, since τ_9 and τ'_9 have the same tMD, therefore a match *ST*(τ_9)→*ST*(τ'_9) is produced. Similarly, we can easily derive matches such as $\tau_{15} \rightarrow \tau'_{13}$, $\tau_{14} \rightarrow \tau'_{12}$, *ST*(τ_{25})→*ST*(τ'_{23}).

Step 4 (Tune existing matches): we analyze the quality of matches and tune some ineffective matches. The quality of matching for a node *N* is analyzed in terms of how much a match of *N* is consistent with matches of children of *N*. For a node *N*, we define the number (*P#*) of positive children's matches, the number (*N#*) of negative children's matches, and the degree of consistency of matching *Con#*(*N*) as follows:

$P\#(N)$ = the number of child C_k satisfying $[P(M(C_k(N))) = M(N)]$
$N\#(N)$ = the number of child C_k satisfying $[P(M(C_k(N))) \neq M(N)]$
$Con\#(N) = P\#(N)$ /the number of children matched $= P\#(N)/(P\#(N)+N\#(N))$

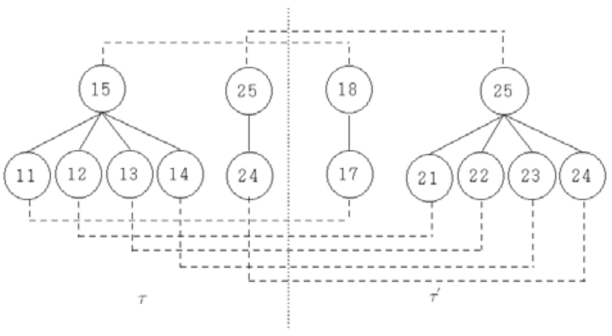
P#(*N*) implies the number of children satisfying that the parent of the matching node of a child of node *N* is equal to the matching node of node *N*, while *N#*(*N*) shows the number of children not satisfying the same condition. *Con#*(*N*) implies the fraction of children contributing positive effects to the current match of *N*. In order to tune ineffective matches, we need to know alternative matches for a node. The list of alternative matches for a node (*lAM*) is defined as follows.

$$lAM\ (N) = \{<K, \text{the number of child } C_i> \mid Label(N)=Label(K) \wedge K \neq M(N) \wedge K=P(M(C_i\ (N))) \text{ for some child } C_i(N)\ \}$$

With *lAM* (*N*), we can find a node that can be matched with node *N*, and also the number of children contributing positive effect to this possible match, which is called the supporting degree for the match. Among the set of pairs represented by *lAM* (*N*), let *fN*(*N*) denote the node with highest supporting degree and *fV*(*N*) the supporting degree. The tuning process begins with traversing τ in post-order. The algorithm is presented in Figure 6.

```
For each node $N$ in $\tau$                    // traverse in post-order
 If $Con\#(N) < 0.5$ then {
   Compute $lAM\ (N)$;
   If $fV(N) > P\#(N) + P\#(fN(N))$ then
     {Revoke the current match of N; Revoke the current match of $fN(N)$; Match $N$ with $fN(N)$;}
   }
```

Fig. 6. Algorithm for Step 4



Data of τ_{15} for tuning

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

Data of τ_{25} for tuning

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

Fig. 7. An example for Step 4(tuning existing matches)

Figure 7 shows parts of two X-trees. Doted lines represent existing matches. Tables show *P#*, *N#*, *Con#* for nodes τ_{15} and τ_{25}. Note that *Con#*(τ_{15}) < 0.5. Since from the *lAM*(τ_{15}), *fN*(τ_{15})=τ'_{25} and *fV*(τ_{15})=3 > P#(τ_{15}) + P#(τ_{25}), we replace τ_{15} 's current match ($\tau_{15} \rightarrow \tau'_{18}$) with new one ($\tau_{15} \rightarrow \tau'_{25}$).

Step 5 (Match remaining identical subtrees with move and copy operations): Among unmatched nodes in τ and τ'. Among them, we try to match identical subtrees, which have not been matched in Step 1, with move and copy operations.

In order to achieve this task, we find unmatched nodes from τ and τ' in breadth-first order traversal, and produce two hash tables, S_Htable for τ and T_Htable for τ'. The entry structure for these hash tables is (tMD value *t*, a list of nodes (LN)) where tMD value is a hash key and the list of nodes (LN) have the same tMD value *t*. The matching process proceeds as follows.

(1) For each entry (h_key$_p$, T_LN$_p$) in T_Htable, look up S_Htable with h_key$_p$. If this lookup fails, then go on to the next entry in T_Htable. (2) In case of the successful lookup, from the list of nodes S_LN$_q$ of the entry looked up in S_Htable and the list of nodes T_LN$_p$, get pair (*N*, *M*) of nodes with the same position in both list and match *ST*(*N*) with *ST*(*M*). Suppose the length of T_LN$_p$ be *lnT* and that of S_LN$_q$ *lnS*. If $lnS \leq lnT$, we match the first m nodes from S_LN$_q$ with the first *lnS* nodes from T_LN$_p$, respectively. Then match the last node in *lnS* with the rest of nodes in T_LN$_p$ using Copy operation. In case of $lnS > lnT$, we match the first *lnT* nodes from S_LN$_q$ with the first *lnT* nodes from T_LN$_p$ using move operation respectively. Then leave the rest of nodes in S_LN$_q$ unmatched.

4 Algorithm Analysis and Experiments

In this section, we present the time complexity of X-tree Diff+ and the result of experiments. $|T|$ denotes the number of nodes in tree *T*. In this paper, since Steps 0-3 are similar to those in X-tree Diff, we just show that the time complexity of Steps 0 – 3 is $O(|T_{old}|+|T_{new}|)$. For the details, refer to the paper [17].

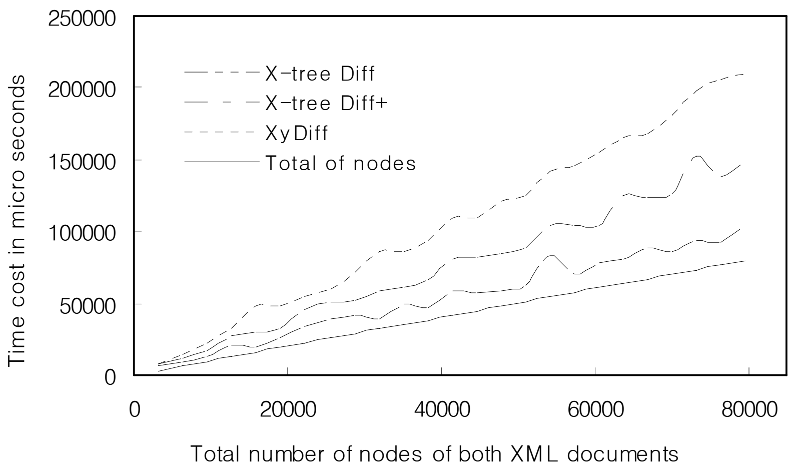


Fig. 8. Measuring the execution time for each algorithm.

In Step 4, we traverse all the nodes in τ in post-order way. When a node *N* is visited, we compute *P#*(*N*), *N#*(*N*), *Con#*(*N*), *lAM*(*N*). The cost of computing these functions, revoking a match and rematching is $O(1)$. All the nodes in τ are counted once as a child during the traversal. Therefore, the cost of Step 4 is also $O(|T_{old}|)$. Step 5 requires traversing τ and τ', and also building hash tables S_Htable and T_Htable. This task costs $O(|T_{old}|+|T_{new}|)$. Matching process using these hash tables costs also $O(|T_{old}|+|T_{new}|)$ in worst case implying all the nodes in both X-trees are considered. The time cost of Step 5 is $O(|T_{old}|+|T_{new}|)$.

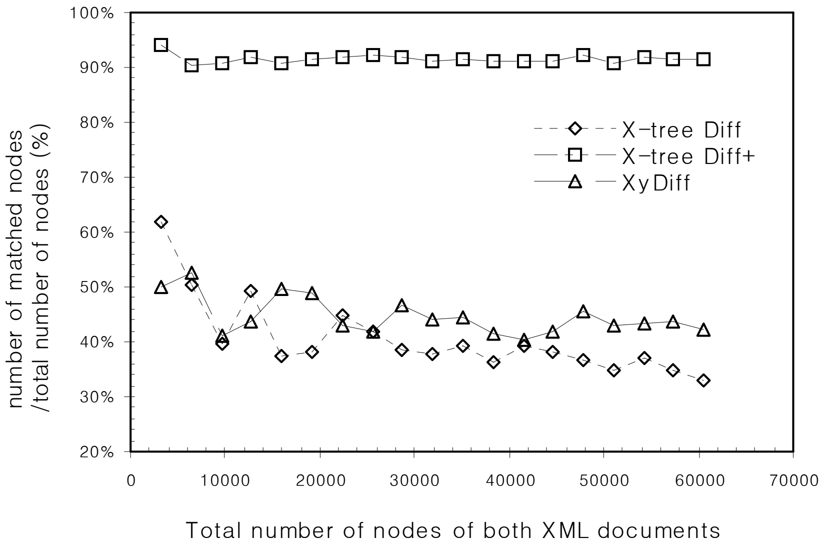


Fig. 9. Matching ratio of each algorithm.

The cost of generating edit scripts is also to $O(|T_{old}|+|T_{new}|)$ [14]. In conclusion, the time cost of X-tree Diff+, even in worst case, is $O(|T_{old}|+|T_{new}|)$. It is equal to that of fastest heuristic algorithms for tree-structured documents. Figure 8 shows the execution time in matching nodes for X-tree Diff, XyDiff, and X-tree Diff+. Y-axis in the graph represents the total execution time except Step 0. The lines for X-tree Diff and X-tree Diff+ are linear, as we analyzed above. In the aspect of efficiency, X-tree Diff is the best, X-tree Diff+ next, XyDiff last. The test data is produced by Synthetic XML Data Generator[15,16]In Figure 9, we show the matching ratio which is the ratio of the number of matched nodes to total number of nodes. The matching ratio of X-tree Diff+ is much higher than those of other two algorithms. It is because of introducing tuning step and copy operation.

5 Conclusion

In this paper, we propose X-tree Diff+, which is fast and produces reasonably good quality of edit scripts. The time complexity of X-tree Diff+ is $O(n)$ in worst case. Its matching ratio is much higher than existing heuristic algorithms. X-tree Diff+

introduces tuning step and copy operation. Even though X-tree Diff+ is heuristic algorithm, X-tree Diff+ uses a systematic approach for tuning; based on the notion of consistency of matching between a parent and its children, analyze the degree of consistency of matching for each node in terms of P#, N#, Con#, then tune ineffective matches. Introducing copy operation provides users convenience. Because users get used to use copy operation in editing software, it is awkward not to support copy operation in existing algorithms. In addition, copy operation increases matching ratio a lot. With systematic tuning step and copy operation, the quality of edit scripts that X-tree Diff+ produces is better.

References

1. 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.
2. S. M. Selkow, "The tree-to-tree editing problem," *Information Proc. Letters*, 6, pp.184-186, 1977.
3. K. Tai, "The tree-to-tree correction problem," *Journal of the ACM*, 26(3), pp.422-433, July 1979.
4. S. Lu, "A tree-to-tree distance and its application to cluster analysis," *IEEE TPAMI*, 1(2), pp.219-224, 1979.
5. J. T. Wang and K. Zhang, "A System for Approximate Tree Matching," *IEEE TKDE, 6(4)*, pp.559-571, August 1994.
6. S. Chawathe and H. G. Molina. "Meaningful Change Detection in Structured Data," *In Proc. of ACM SIGMOD '97*, pp.26-37, 1997.
7. S. Chawathe, "Comparing Hierarchical Data in External Memory," *In Proc. of the 25th VLDB Conf.*, pp.90-101, 1999.
8. 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.
9. Curbera and D. A. Epstein, "Fast Difference and Update of XML Documents," *XTech '99*, San Jose, March 1999.
10. G. Cobéna, S. Abiteboul and A. Marian, "Detecting Changes in XML Documents," *the 18th ICDE*, 2002.
11. Y. Wang, D. J. DeWitt, J. Y. Cai, "X-Diff : An Effective Change Detection Algorithm for XML Documents," *the 19th ICDE*, 2003.
12. 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.
13. R. Rivest, "The MD4 Message-Digest Algorithm," *MIT and RSA Data Security, Inc.*, April 1992.
14. D. A. Kim and S. K. Lee, "Efficient Change Detection in Tree-Structured Data," In Human.Society@Internet Conf. 2003, pp.675-681, 2003.
15. 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.
16. NIAGARA Query Engine, http://www.cs.wisc.edu/niagara/ .
17. D. A. Kim, “Change Detection and Management in XML Documents” Ph.D. thesis, Dankook University, Korea, 2005.