# KF-Diff+: Highly Efficient Change Detection Algorithm for XML Documents

Haiyuan Xu, Quanyuan Wu, Huaimin Wang, Guogui Yang, and Yan Jia

Laboratory 613, School of Computer Science, National University of Defense Technology, Changsha, Hunan, P.R. China
hyxu@nudt.edu.cn

**Abstract.** Most previous work in change detection on XML documents used the ordered tree, with the best complexity of O(nlogn), where n is the size of the document. The best algorithm we had ever known for unordered model achieves polynomial time in complexity. In this paper, we propose a highly efficient algorithm named KF-Diff+. The key property of our algorithm is that the algorithm transforms the traditional tree-to-tree correction into the comparing of the key trees which are substantially label trees without duplicate paths with the complexity of O(n), where n is the number of nodes in the trees. In addition, KF-Diff+ is tailored to both ordered trees and unordered trees. Experiment shows that KF-Diff+ can handle XML documents at extreme speed.

## 1 Introduction

Data publication on the web is constantly increasing. The web constitutes the largest body of information accessible to any individual throughout the history of humanity and keeps growing at a healthy pace [14]. It should be pointed that change monitoring is becoming very popular on the web. Users are often not only interested in the current values of documents and query answers but also in changes. They want to see changes as information that can be used to learn about the evolution of the web.

One of the key challenges for building a large-scale web based change monitoring system is that it must scale to large number of sources and subscriptions, since that in the Internet environment, where huge volumes of input data and large numbers of users are typical, efficiency and scalability are key concerns. In our XFDS project [12], a partly implemented prototype system designed to monitor the fetching of millions of XML [13] documents per day where supporting millions of subscriptions, we are lead to compute the difference between the millions of documents obtained each day and previous version of these documents. This motivates the study of an extremely efficient change detection algorithm for tree data. Usually, the change detection algorithm will determine whether or not the two versions are identical. If not, then tries to match each element in the old version with every one in the new version to know the difference of the two versions. Since we focus on deltas, as opposed to other approaches that might focus on snapshots or object history, and delta data is usually much smaller than the original data, query evaluation will be much faster.

Since XML documents can be represented as trees, the change detection problem is related to the problem of change detection on trees. Algorithms to compute the difference between trees can be divided into two categories depending on whether they deal with ordered or unordered trees. An ordered tree is one in which both the ancestor relationship and the left-to-right ordering among siblings are significant. An unordered tree is one in which only ancestor relationships are significant, where the left-to-right order among siblings is not significant. Change detection on unordered trees has been shown to be NP-Complete in general case and is substantially harder than that on ordered trees. For unordered trees, we term two trees isomorphic if they are identical except for the orders among siblings. We considers two trees are equivalent if they are isomorphic. In this paper, we introduce some notions (e.g., key path), which are used to compute the difference between two trees. Based on these notions, we propose a highly efficient algorithm named KF-Diff+. The key property of our algorithm is that the algorithm transforms the traditional tree-to-tree correction into the comparing of the key trees which are substantially label trees without duplicate paths. Thus, our algorithm achieves high efficiency with the complexity of O(n), where n is the number of nodes in the trees. In addition, KF-Diff+ is tailored to both ordered trees and unordered trees, which is also a significant difference compared with previous work. KF-Diff+ has two kinds of forms named algorithm 1 and algorithm 2 respectively. Algorithm 1 has the complexity of O(n) by using hash tables, which usually consumes much more time than algorithm 2. Algorithm 2 usually achieves better performance with the complexity estimated of $O(n\log\lambda)$, where $\lambda$ is the average out-degree of nodes in the document. Experiments shows that KF-Diff+ can handle XML documents at extreme speed, which is significant to the large scale applications.

The remainder of the paper is organized as follows. Related work is contained in Sect. 2. In Sect. 3, we formulate the problem and give an overview of our approach. Sect. 4 presents the details of KF-Diff+ with complexity analysis. Sect. 5 gives some preliminary performance results. Our conclusions and future research directions are contained in Sect. 7.

## 2 Related Work

Some previous work in change detection has focused on computing differences between flat files, e.g. GNU diff and CVS. They cannot be generalized to handle structured data because they do not understand the hierarchical structure information contained in such data sets. The AT&T Internet Difference Engine [6] uses HtmlDiff [1] to determine the differences between two HTML pages. This method cannot be applied to XML documents because markups in XML data provide context, and contents within different markups cannot be matched. Since XML documents can be represented as trees, it is a natural idea to utilize tree-to-tree correction techniques to detect changes in XML documents. Zhang and Shasha proposed a fast algorithm to find the minimum cost editing distance between two ordered labeled trees [9]. Given two ordered trees $T_1$ and $T_2$, in which each node has an associated label, their algorithm finds an optimal edit script in time $O(|T_1| \times |T_2| \times \min\{depth(T_1), leaves(T_1)\} \times \min\{depth(T_2), leaves(T_2)\})$, which is the best known result for the general tree-to-tree correction problem.

Recently, Sun released an XML specific tool named DiffMK [4] that computes the difference between two XML documents. This tool is based on the unix standard diff algorithm, and uses a list description of the XML document, thus losing the benefit of tree structure of XML. [2, 5, 3, 11, 8] deal with the problem on hierarchically structured documents. However, LaDiff [2] is limited by its assumption and not guaranteed to generate the optimal result. MH-Diff [5] provides an efficient heuristic solution based on transforming the problem to the edge cover problem, with a worst case cost in $O(n^2 \log n)$, where n is the total number of nodes. XMLTreeDiff [3] use DOMHash [7] and Zhang's algorithm [9]. Since the former conflicts with the later, this method may not generate an optimal result. [11] proposed XyDiff, an algorithm for detecting changes in XML documents. This algorithm achieves O(nlogn) complexity in execution time where n is the size of the document and generates fairly good results in many cases. However, XyDiff cannot guarantee any form of optimal or near-optimal result because of the greedy rules used in the algorithm. The major difference between [8] and [2, 5, 3, 11] is that the former is designed to handle unordered trees, where the later for ordered trees. [8] presents an algorithm named X-Diff for computing the difference between two versions of an XML document, which achieves polynomial time in complexity. Since X-Diff can only reduce the size of tree by removing equivalent second-level subtrees, the filtering is not efficient. Besides, in some case, X-Diff may not satisfy uses' needs. In this paper, we propose a highly efficient algorithm named KF-Diff+. The key property of our algorithm is that the algorithm transforms the traditional tree-to-tree correction into the comparing of the key trees which are substantially label trees without duplicate paths.

## 3 Preliminaries

### 3.1 Tree Representation of XML Documents

**Definition 1. A**ssume a countably infinite set E of element labels (tags), a countably infinite set A of attribute names, and a symbol S indicating text (e.g., PCDATA in XML ). Assume that E, A and {S} are pairwise disjoint. An XML (document) tree [10] T is defined to be (V, lab, Ele, att, val, r), where V is a set of *vertices* (*nodes*); lab is a function from V to $E \cup A \cup \{S\}$; Ele is a partial function from V to sequences of V vertices such that for any $v \in V$, if Ele(v) is defined then $lab(v) \in E$; att is a partial function from $V \times A$ to V such that for any $v \in V$ and $l \in A$, if $att(v,l) = v'$ then $lab(v) \in E$ and $lab(v') = l$; val is a partial function from V to string values such that for any node $v \in V$, val(v) is a string iff either lab(v) = S or $lab(v) \in A$ ; r is a distinguished vertex in V and is called the root of T. Without loss of generality, assume lab(r) = r , and that there is a unique node in T labeled r.

**Definition 2.** A (simple ) *path* is a sequence of node labels, syntactically defined as follows: $\rho \equiv \varepsilon \mid l . \rho$ where $\varepsilon$ is the empty path, node label $l \in E \cup A \cup \{S\}$, and "." is the concatenation operator. Intuitively, a path represents the sequence of node labels in a parent-child path in an XML tree. More precisely, let T be an XML tree, $v_1$, $v_2$ be nodes in T and $\rho$ be a path. We say that there is a path from $v_1$ to $v_2$, denoted by $T \models \rho(v_1, v_2)$, if there is a parent-child path from $v_1$ to $v_2$ whose sequence of node labels is $\rho$.

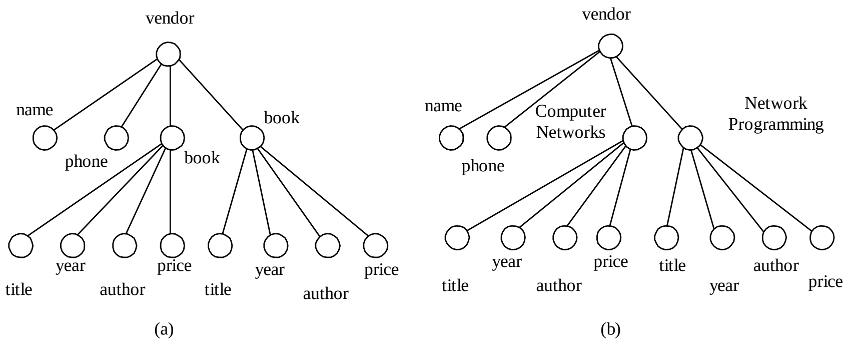

**Fig. 1.** (a) Tree representation for an XML document, (b) An example of Key tree

### 3.2 Key Field and Key Path

As mentioned before, many previous work utilize tree-to-tree correction techniques to detect changes in XML documents. However it is still a problem that how to compare the nodes which have the same path. For instance, as shown in Figure 1(a), there are two book elements with the same path in the tree. For unordered model, a naïve way is to compare them each other, which is obviously not efficient. For ordered model, the left-to-right ordering can help to discriminate among these nodes. However, if these nodes are compared directly according to the ordering, the quality of the result can not be guaranteed. So, can we discriminate those nodes in the instinctive features of the XML document? To answer the question, we introduce the notion of the key constraint [10] (see Sect. 3.3) in order to support such a discrimination. That is, we want to substitute the value of the key constraint, denoted by *key value*, for the label of the nodes. For example, suppose a vendor has many books which each has an unique name, thus we can use the name of book as the value of the key constraint corresponding to the path "vendor/book". Suppose the name of the two books are "Network Programming" and "Computer Networks" respectively, we will see the tree changed in Fig. 1(b). It should be pointed that we need not to change all the labels of the elements. It is another question that what's kind of nodes should be changed. To answer the question, we introduce the notion of Multi-Instance node. A set of the Multi-Instance nodes is substantially the set of nodes that have the same father node and the same label.

**Definition 3.** A set of *Multi-Instance* nodes, denoted by MI, is defined as follows: Suppose v is a node in a tree T, If there is a node v′ in T, $v \neq v'$, lab(v) = lab(v′), and there is another node n in T, and have $v' \in Ele(n) \wedge v \in Ele(n)$ , then MI(v) = $\{ v' | lab(v) = lab(v'), \exists\ n \in V\ ( v' \in Ele(n) \wedge v \in Ele(n))\}$, otherwise MI(v) = $\phi$, where $\phi$ denotes empty set. That is, MI is meaningful only when it has at least two satisfied nodes.

To explain the relation between MI and the set of nodes which have the same path, we give the following theorem.

**Theorem 1.** Suppose v, v′ are different nodes with the same path ρ in T, $(e_1, e_2, \ldots, e_k)$ are the nodes from r to v along ρ, and $(e_1', e_2', \ldots, e_k')$ are the nodes from r to v′ along ρ (obviously, $e_1 = e_1' = r$, $e_k = v$ and $e_k' = v'$), we have that there is one and only one MI, denoted by S, which makes the following true:

$(\exists\ i \in [1, k])( e_i \in S \wedge e_i' \in S \wedge e_{i-1} = e_{i-1}')$.

Proof is omitted here. Theorem 1 shows that for any two nodes that have same path, we can find such a MI, and then substitute the labels of two nodes (such as $e_i$ and $e_i'$ described above, or two book elements in Fig. 1(a)) for the different values. Intuitively, after we handle all the MIs in the tree, it seems there is no elements that have same path (as shown in Figure 1(b)). We will proof it in the theorem 2. Since the value used to take the place of the label of node v in the MI comes from the subtree rooted at v, the relative path (from v to the leaf node which contain the value) is called a *key field*. We next present the definition of the *key field*.

**Definition 4.** A MI is substantially a set that consists of the sibling nodes with the same label. To describe the difference of the nodes in the MI, we introduce the notion of *key field*, denoted by KF, which is defined as follows: If MI (v) $\neq \phi$, then KF (v) = $\{\rho \mid \rho$ is a relative path expression, and $\forall ( e_1, e_2 \in MI(v))((e_1 \neq e_2) \rightarrow (\exists (k_1, k_2 \in V) (T \models \rho(e_1, k_1) \wedge T \models \rho(e_2, k_2) \wedge val(k_1) \neq val(k_2))))\}$. Obviously, ρ can be ε, in which case that v is a leaf node. It should be pointed that the definition above is a simple form. In some case, there may be no such a satisfied key field. To solve the problem, we give an extended definition described as following: KF (v) = {P | P is a set of relative paths from v to the leaf nodes in the subtree rooted at v, and have

$$\forall ( e_1, e_2 \in MI(v))((e_1 \neq e_2) \rightarrow NOT( \bigwedge_{1 \le i \le k} (\exists (k_1, k_2 \in V) (\rho_i \in P \wedge T \models \rho_i (e_1, k_1) \wedge T \models \rho_i (e_2, k_2) \wedge val(k_1) = val(k_2)))))$$

where k is the total number of paths in P. The extended definition is tailored to almost all kinds of XML documents because in the case that there is no such an extended key field, there must be two nodes in the MI which are value equal (see Sect. 3.3), where such nodes need not to be discriminated at all. In this case, we can neglect those nodes which are value equal because they do not change the correctness of the algorithm. Due to the space limitation, we only discuss the simple form of the definition in the paper, which is usually enough in practice.

As mentioned above, KF is a notion relates to MI. For each node v in the MI, it shows where to find the key value in the subtree rooted at v. The key property of KF is that given a MI and associated KF, any two different nodes in the MI should not have the same value corresponding to the KF. As mention before, some elements's labels should be changed in the tree, where others not. Thus, in the key tree, as shown in Figure 1(b), the notion of the label is changed. We next give the notion of *key label*. Intuitively, the *key label* of an unchanged element is exactly the original label of the element, where the *key label* of an changed element is not. Here is the definition.

**Definition 5.** The notion of *key label* of node v, denoted by KLab, is defined as follows: 1) If MI (v) = $\phi$, then KLab(v) = lab(v) ; 2) Otherwise, suppose p is one of the paths of KF(v) and $T \models p(v, k)$, then KLab(v) = val(k). We also define *key path*, denoted by KPath, as follows: KPath(v) = KLab($e_1$) … KLab($e_k$) . KLab(v), where ($e_1$, $e_2$, . . ., $e_k$, v) are nodes from r to v. The tree consists of KLabs is denoted by *key tree*.

Notice that KF (v) may has many satisfied paths, in theory, we can use any of them to obtain a key value. In practice, we utilize some criterions to select better key values. For example, we much prefer to select the key value that is less likely be modified. The detail is omitted here. In this paper, to be more precisely presented, definition 5 is based on assumption 1.

**Assumption 1.** Given any $v \in V$ and MI (v) $\neq \phi$, for any different sibling element $v'$ of v, we have KLab(v) $\neq$ KLab($v'$).

Observe that this assumption is usually satisfied. In addition, in our implementation, we have some special ways to deal with the case encountered that the assumption is not satisfied. The detail is out of the scope of the paper.

Now, we transform the tree-to-tree correction into the comparing of the key trees. We expect that the key path is unique in the key tree. So we give the following theorem.

**Theorem 2.** Suppose $e_1$, $e_2$ are nodes in the T, KPath($e_1$) = KPath($e_2$) iff $e_1 = e_2$.

Here we give a brief proof for Theorem 2.

Proof: $\Leftarrow$ is obvious.

$\Rightarrow$ Given KPath($e_1$) = KPath($e_2$), then by definition 5, there must exist nodes ( $x_1$, $x_2$, …, $x_k$, $e_1$) from r to $e_1$, and ( $y_1$, $y_2$, …, $y_k$, $e_2$) from r to $e_2$, and KLab($x_1$) … KLab($x_k$) . KLab($e_1$) = KLab($y_1$) … KLab($y_k$) . KLab($e_2$), thus we have KLab($x_i$) = KLab($y_i$) for all $i \in [1, k]$ , and KLab($e_1$) = KLab($e_2$). Since $x_1$, $y_1$ are both the root of T, we have $x_1 = y_1 = r$. Now we consider the case i = 2, suppose $x_2 \neq y_2$, thus we have that $x_2$ and $y_2$ are sibling nodes. Since KLab($x_2$) = KLab($y_2$), this contradicts the assumption 1. Similarly, we can proof for all $i \in [1, k]$, we have $x_i = y_i$, and similarly, we have $e_1 = e_2$. Theorem 2 shows that there is no case that two different elements have same key path in the key tree, which is the foundation of our algorithm.

Now the question is how to obtain the key value of MI. To answer the question, we need to introduce the notion of key constraint. First, we give the notion of value equal which is central to a definition of *key constraint*. In relational databases, this is not a problem: one needs only to compare values of atomic types when checking the satisfaction of a key. An XML tree has a hierarchical structure and it is no longer trivial to compare the values of two XML trees (subtrees). Intuitively, we need a definition of equality on the values of XML trees such that if two trees are *value equal*, then the two XML documents represented by the two trees are the same. Here is the definition of *value equal*.

## 3.3 Selection of the Key Field

**Definition 6.** We say that v and v′ are *value equal* [10] denoted by $v =_v v'$, iff the following conditions are satisfied: 1) lab(v) = lab(v′); 2) if lab(v) = S or lab(v) $\in$ A then val (v) = val (v′); 3) if lab(v)$\in$ E, then for any l$\in$ A, att(v, l) is defined iff att(v′, l) is defined, and val (att(v, l)) = val (att(v′, l)); if Ele(v) = $[e_1, e_2, \ldots, e_k]$, then Ele(v′) = $[e_1', e_2', \ldots, e_k']$ and for all i$\in$ [1, k], $e_i =_v e_i'$. In short, $v =_v v'$ iff their substrees are isomorphic by an isomorphism that is the identity on string values.

In relational databases, to define a key we specify the name of a relation (a set of tuples) and a set of attributes of the relation that uniquely identifies tuples in the relation. Along the same lines, to define a *key constraint* φ for XML data we specify a form $(Q,\{P_1,\ldots, P_k\})$, where Q is a path called the target path of φ ; $P_1, \ldots, P_k$ are path expressions called the feature paths of φ with k≥1. We use [[Q]] to denote the set of nodes in T that is reachable by following Q from the root r, and use n[[P]] to denote the set of nodes reachable form n by following P. An XML tree T satisfies φ, denoted by $T \models \varphi$, iff $\forall e_1, e_2 \in [[Q]] \; (\bigwedge_{1\le i\le k} \exists x \in e_1[[P_i]] \; \exists y \in e_2[[P_i]](x =_v y) \longrightarrow e_1 = e_2)$

There are two forms of *keys* are important for hierarchically structured data. The first is *absolute key*. Similar to relational database keys, an *absolute key* identifies a unique node x in [[Q]] with the values of nodes in the subtree rooted at x. The second is *relative key*, which is analogous to a key for a weak entity in a relational database. In this paper, we only consider the former. Now, it is time for answering the question. Intuitively, the *feature paths* in the *key constraint* are the satisfied paths in the *key field*. It should be pointed that the condition of the *feature paths* are stronger than that of the *key field*. Thus, we relax the satisfaction of the keys by follows: given *key* $\varphi = \{ Q, \{P_1,\ldots, P_k\}\}$, we say $T \models \varphi$, iff for each MI that is reachable by following Q from the root r, $\forall e_1, e_2 \in \mathrm{MI} \; (\bigwedge_{1\le i\le k} \exists x \in e_1[[P_i]] \; \exists y \in e_2[[P_i]](x =_v y) \longrightarrow e_1 = e_2)$. This means that we only discuss the satisfaction inside the MI, which can significantly reduce the cost of finding keys. Due to the space limitation, we only consider the *keys* that have just one feature path in this paper.

Let ∑ be a finite set of *key constraints*, given a MI, corresponding *key field* can be obtained as one of the follows: i) utilizing the ID attribute in the DTD of the documents; ii) utilizing the index of *keys*; iii) by experience. For instance, some labels such as “title”, “name”, “id”, “ssn”, etc, usually fit for key field; iv) computing directly. Since the number of nodes in the MI usually is not very large, the time cost will not be very much. In addition, usually we can get keys before computing the difference between trees, so, the cost of finding keys does not change the performance of the algorithm in most case.

## 4 Change Detection with KF-Diff+

In this section we propose KF-Diff+, a highly efficient algorithm which need not to compute the hash signatures of all the nodes in the documents. As mentioned before, the key property of KF-Diff+ is that the algorithm transforms the traditional tree-to-tree correction into the comparing of the *key trees*. Since any two different nodes in the key tree should not have the same *key path*, the comparing between two trees will be more efficient.

There are two steps in KF-Diff+ as follows: 1). Parsing. KF-Diff+ parses $DOC_1$ and $DOC_2$ into trees $T_1$ and $T_2$. During the parsing process, KF-Diff+ will find out all MIs and compute the *key paths* associated. 2).Matching. The goal of this step is to find a minimum-cost matching between $T_1$ and $T_2$. Due to the space limitation, the generating of the edit scripts [15] is omitted here.

```
Input:  DOC_1, DOC_2
Output:  T_1, T_2 , HT[ ] ( HT[i] is a hash table)
Initialize:  T_1 = T_2 = NULL , HS[i] =NULL, i∈ (2, DEPTH),
                    where DEPTH is the maximal depth of the DOC_1.
for each element v in the DOC_1 do
   Assign v a ID denoted by EID(v);  Construct T_1;
   Find out whether the node belongs to a MI, and if so,
      look for the associated key field and compute the KLab(v);
for each element v in the DOC_2 do
   Assign v EID(v);  Construct T_2;
   Find out whether the node belongs to a MI, and if so,
      look for the associated key field and compute the KLab(v).
   if v is not the root of T_2 Put <KPath(v), EID(v)> into HT [depth(v)]
return T_1, T_2 , HT[ ]
```

**Fig. 2.** Preprocessing phase of Algorithm 1

### 4.1 Algorithm 1

**Parsing.** The parsing phase, as shown in Fig. 2, is the preprocessing step in KF-Diff+. Two input XML documents, $DOC_1$ and $DOC_2$, are parsed into trees first. Since DOM tree contains much unnecessary information and consumes more memory, we just construct a simple tree by using SAX programming interface. Notice that during the parsing process, KF-Diff+ will find out all MIs and compute the key paths associated, which will be used directly by the later step.

**Matching.** The algorithm is shown in Fig. 3. According the theorem 2, KPath(x) $\neq$ KPath(y) iff $x \neq y$. Thus, in the matching algorithm, we compare $T_1$ with $T_2$ directly by using key paths of nodes.

```
Input: $T_1$, $T_2$, HT[ ] (obtained by previous step)
Output: DeletedNodes, InsertedNodes, UpdatedNodes
Initialize: Suppose = Root($T_1$), $r_2$ = Root($T_2$), $TempSet_1$ = $TempSet_2$ = {},
            DeletedNodes = InsertedNodes = UpdatedNodes = {};
First Step: if lab($r_1$) $\neq$ lab($r_2$)      return NULL
                else $TempSet_1$ = Children( $r_1$), $TempSet_2$ = Children( $r_2$)
   /* Children(v) is the function that returns all the child nodes of v*/
Second Step:  Call MatchFunc($TempSet_1$, $TempSet_2$)
return DeletedNodes, InsertedNodes, UpdatedNodes
MatchFunc($TempSet_1$, $TempSet_2$)
{  for each element v in $TempSet_1$ do
        Look for KPath(v) in the hash table HT[depth(v)]
          if the result is NULL move v to DeletedNodes
          else let v′ denotes the corresponding node in $T_2$ we get
             Remove v′ from $TempSet_2$
             if v is a leaf node
                if val(v) $\neq$ val(v′)   add <v, val(v′) > to UpdatedNodes
             else call MatchFunc( Children(v), Children(v′))
        if v is the last element in $TempSet_1$
          Add all unprocessed nodes in $TempSet_2$ to InsertedNodes
}
```

**Fig. 3.** Matching of Algorithm 1

**Algorithm Analysis.** Obviously, the complexity of parsing is O($|T_1|$ + $|T_2|$), and the complexity of matching is also O($|T_1|$ + $|T_2|$), without considering the hashing cost. Thus the complexity of algorithm 1 is O(n), where n is the total number of the nodes in the trees.

## 4.2 Algorithm 2

**Parsing.** The parsing is similar to the one of algorithm 1. The difference is algorithm 2 do not use hash tables.

**Matching.** The algorithm is shown in Fig. 4. Similarly, algorithm 2 compares $T_1$ with $T_2$ directly by using key paths. The difference is that algorithm use set oriented techniques instead of hash tables.

**Algorithm Analysis.** Similarly, the complexity of parsing is O($|T_1|$ + $|T_2|$). Suppose there are N matching pairs between $T_1$ and $T_2$, thus the cost of matching is bounded by $\sum_{i=1}^{N} O(\max\{\deg(x_i),\deg(y_i)\}\times\log(\max\{\deg(x_i),\deg(y_i)\}))$, where deg(x) denote the out-degree of node x in the trees. Consider the case that every element has the same out-degree, denoted by λ. Thus we have that the number of non-leaf elements is

```
Input:  $T_1$, $T_2$ (obtained by previous step)
Output:  DeletedNodes, InsertedNodes, UpdatedNodes
Initialize: Suppose $r_1$ = Root($T_1$), $r_2$ = Root($T_2$), $TempSet_1$ = $TempSet_2$ = M = {},
            DeletedNodes = InsertedNodes = UpdatedNodes = {};
First Step:   if lab($r_1$) $\neq$ lab($r_2$)  return NULL;
              else $TempSet_1$ = Children($r_1$), $TempSet_2$ = Children($r_2$);
Second Step:  Call MatchFunc($TempSet_1$, $TempSet_2$);
return DeletedNodes, InsertedNodes, UpdatedNodes
MatchFunc($TempSet_1$, $TempSet_2$)
{  TempSet = {} /* local variable */
   Sort the elements in $TempSet_1$ according KLab;
   Sort the elements in $TempSet_2$ according KLab;
   Compare $TempSet_1$ with $TempSet_2$ according KLab; /* sort merge */
   for each matching pair <v, v′> do
      /* v belongs to $TempSet_1$, v′ belongs to $TempSet_2$ */
      Remove v from $TempSet_1$, Remove v′ from $TempSet_2$
      Add <v, v′> to M
         if v is a leaf node
            if val(v) $\neq$ val(v′) add <v, val(v′) > to UpdatedNodes
         else add v to TempSet
   Add remainder nodes in $TempSet_1$ to DeletedNodes
   Add remainder nodes in $TempSet_2$ to InsertedNodes
   For each element v in TempSet do
      Get corresponding v′ in M
      Call MatchFunc ( Children(v), Children(v′))
   }
```

**Fig. 4.** Matching of Algorithm 2

less than $(1/\lambda) \times n$, where n is the total number of the nodes in the trees. Since the out-degree of the leaf node is zero, the cost of matching is less than $O((1/\lambda) \times n \times \lambda\log\lambda) = O(n \log\lambda)$. Note that $O(n \log\lambda)$ is only an estimate of the complexity. In the case that $\lambda \leq 2$, the complexity of the matching is O(n). In the worst case that all other nodes are the child nodes of the root, that is $\lambda = n-1$, the complexity of algorithm 2 is $O((n-1)\log(n-1))$.

## 4.3 KF-Diff+ for Ordered Trees

The algorithms mentioned above can easily be tailored to handle ordered trees which the left-to-right ordering among siblings are significant. The main differences are as follows. 1) Ordering of the nodes are considered; 2) Move operations are considered. It should be pointed that the algorithms presented in this paper only support limited detection of move operations. That is to say, we can only detect the case that nodes are moved within the sibling nodes, and in other case, such a move operation will be detected into one delete operation plus one insert operation. Notice that in practice, the most move operations occur within the sibling nodes. Of course, we can adapt our algorithms to detect more move operations. However, it is beyond the scope of the present paper.

## 5 Implementation and Performance

### 5.1 Implementation

We have implemented KF-Diff+ in C and C++. It takes two XML documents as input. We implement a workload generator that, according to a workload specification, emits XML documents modified to the system. The workload generation task ran as a separate process on the another PC. The parameters(e.g., average number of change rate) may be set in the workload specification. We implement a faster XML parser based on the SAX interface. To reduce the time cost, we store the logic trees instead of the original XML documents. Thus, we need not to parse the old document and compute the key tree again. Since our data structure is very simple, it nearly does not take additional space cost. It should be pointed that the structure contains some information associated with key paths in the tree which can be directly used by the change detecting algorithm, so as to save the time for recomputing. To reduce the cost of disk I/O, we implement some cache strategies in order to preload the data of the previous version. For more detail about logical storage of the document and cache strategies, please refer to [12], a partly implemented prototype system based on the KF-Diff+. Notice that KF-Diff+ does not require two XML documents to conform the same schema.

### 5.2 Performance

In this section, we study the performance of KF-Diff+. Since XyDiff is the best algorithm for ordered tree that we had ever known, and algorithms for unordered trees usually perform worse than those for ordered trees, we compare KF-Diff+ with XyDiff in this paper. Due to the space limitation, we only show some preliminary performance results.

We use a PC with a PIII 850 CPU and 512M RAM operating under Linux. Let C denotes the average change ratio of the documents, $\lambda$ denotes the average out-degree of the documents.

First, we evaluate the execution time of the three algorithms, XyDiff, algorithm 1 and algorithm 2, on documents of different sizes. In Fig. 5(a), given $\lambda = 3.1$, $C = 35\%$, the figure shows that execution times of the three algorithms are almost linear in the size of the document. Notice that algorithm 2 is much faster than algorithm 1, although the complexity of algorithm 2 is higher than algorithm 1, where algorithm 1 is faster than XyDiff. The reason mainly lies in that algorithm 1 uses hash table which consumes more time.

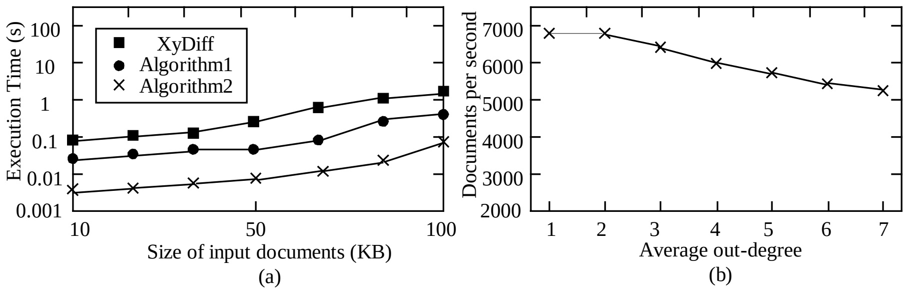

**Fig. 5.** (a) Execution time of the algorithms, (b) Documents per second with varying out-degree

Given C = 50%, workload parameters are fixed as follows: Each generated XML document has a maximal depth of 16, and average depth of 4.4 and an average number of 29.4 elements. Fig. 5(b) shows that how many documents can be handled per second by algorithm 2, as the average out-degree of the documents varies. Observe that the absolute execution time of the algorithm increase significantly when the average out-degree of the document increases. This is consistent with our analyse which demonstrates that the complexity of the algorithm 2 increases as the out-degree increases. Notice also that the algorithm 2 is highly efficient as the out-degree is small, and can process nearly 7000 documents per second, which is very important for large scale applications.

Given λ = 4.1, Fig. 6(a) shows the average processing time for various change rates of the XML documents. This Figure shows that: the execution time almost has no change when change ratio increases. This is because KF-Diff+ does not filter the equivalent part of the two versions.

Given λ = 3.5, C = 80%, Fig. 6(b) shows the processing time per document of KF-Diff+ when the ratio of the modify operation varies. The result shows that KF-Diff+ is effected by the percentage of the modify operations. Observe that given a fixed change rate, the more insert and delete operations are in the delta, the higher the performance is. This is because that for the nodes inserted or deleted, we do not need to compare their descendants.

Given λ = 4.1, Fig. 7 shows that the result obtained by running KF-Diff+ (algorithm 1 and algorithm 2 generate exactly the same result), is close to the optimal difference. One reason that KF-Diff+ generates non-optimal results is that it can not detect the move operations that do not occur within the sibling nodes, in which case each move operation can be detected into one delete operation plus one insert operation. Notice that in practice, such kind of move operations are not occur often. Another reason is that the key fields may be modified (although not occur often), in which case the modify operation can be detected into one delete operation plus one insert operation. However we can adapt the system to minimize the problem in practice.

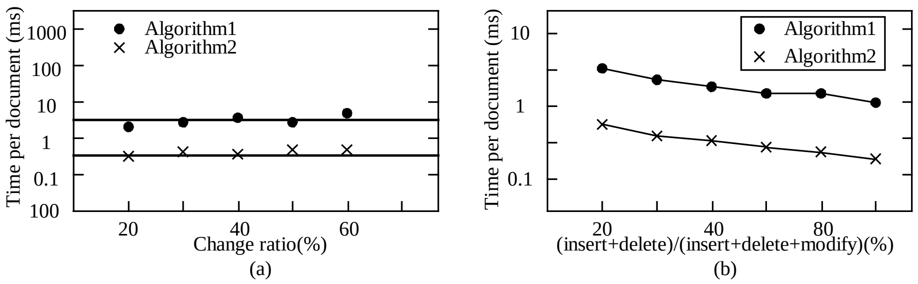

**Fig. 6.** (a) Varying change ratio (b) Varying ratio of insert and delete operation

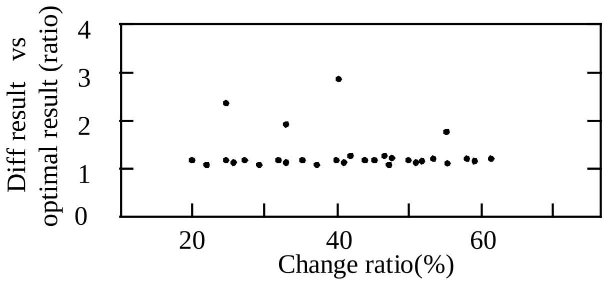

**Fig. 7.** Quality of diff result

### 5.3 Contribution of Key Path

The main contribution of the paper is that we transforms the traditional tree-to-tree correction into the comparing of the key trees, which are substantially label trees without duplicate paths, so that our algorithm achieves high efficiency with linear complexity. In addition, KF-Diff+ is tailored to both ordered trees and unordered trees, which is also a significant difference compared with previous work.

## 6 Summary and Future Work

KF-Diff+ is motivated by the problem of efficiently detecting changes to XML documents on the web. Most previous work in change detection on XML or other hierarchically structured data used the ordered trees. In this paper, we propose a highly efficient algorithm named KF-Diff+. The key property of our algorithm is that the algorithm transforms the traditional tree-to-tree correction into the comparing of the key trees which are substantially label trees without duplicate paths. Thus, our algorithm achieves high efficiency with the complexity of O(n), where n is the total number of nodes. In addition, KF-Diff+ is tailored to both ordered trees and unordered trees, which is also a significant difference compared with previous work. We analyze the algorithm, and give some preliminary performance results.

Many issues may be further investigated. For instance, we plan to continue the study of the feature of the key fields on real web data. Other aspects of the actual implementation could be improved for a different trade-off in quality over performance.

**Acknowledgements.** We would like to thank Yanli Diao, Brain Cooper, Prasan Roy, and Louiqa Raschid for the valuable comments they provided.

## References

1. Berk, E.: HtmlDiff: A Differencing Tool for HTML Documents. Student Project, Princeton University
2. Chawathe, S., Rajaraman, A. Garcia-Molina, H.: Change Detection in Hierarchically Structured Information. Proceedings of the ACM SIGMOD International Conference on Management of Data, Montreal, June 1996.
3. Curbera, F. P.: Fast Difference and Update of XML Documents. XTech'99, San Jose, March 1999.
4. Microsystems, S.: Making all the difference. http://www.sun.com/xml/developers/diffmk/.
5. Chawathe, S., Garcia-Molina, H.: Meaningful change detection in structured data. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Tuscon, Arizona, May 1997.
6. Douglis, F., Ball,T., Chen,Y. F., Koutsofios, E.: The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web. World Wide Web, 1(1): 27-44, January 1998.
7. Maruyama, H., Tamura, K., Uramoto, R.: Digest values for DOM (DOMHash) proposal. IBM Tokyo Research Laboratory, http://www.trl.ibm.co.jp/projects/xml/domhash.htm, 1998.
8. Wang, Y., DeWitt, D. J., Cai, J.: X-Diff: A Fast Change Detection Algorithm for XML Documents. http://www.cs.wisc.edu/~yuanwang/xdiff.html.
9. Zhang , K., Shasha, D.: Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. SIAM Journal of Computing, 18(6): 1245-1262, 1989.
10. Fan, W., Schwenzer, P., Wu, K.: Keys with Upward Wildcards for XML. Database and Expert Systems Applications, 657-667, 2001.
11. Cobéna, G., Abiteboul, S., Marian, A.: Detecting Changes in XML Documents. ICDE, Feb, 2002.
12. Xu, H., Wu, Q., Wang, H., Yang, G., Jia, Y.: XFDS: Efficient Monitoring and Filtering of XML Information on the Web. submitted to publication, 2002.
13. World Wide Consortium. Extensible markup language (xml) 1.0. http://www.w3.org/TR/REC-xml, 2000.
14. The internet archive. http://www.archive.org/
15. Zhang, K.: A New Editing based Distance between Unordered Labeled Trees. Combinatorial Pattern Matching, 1: 254 – 265, 1993.