A Semantical Change Detection Algorithm for XML

Rodrigo Cordeiro dos Santos
Universidade Federal do Parana, Brazil
rodrigosantos@celepar.pr.gov.br

Carmem Hara
Universidade Federal do Parana, Brazil
carmem@inf.ufpr.br

Abstract

XML diff algorithms proposed in the literature have focused on the structural analysis of the document. When XML is used for data exchange, or when different versions of a document are downloaded periodically, a matching process based on keys defined on the document can generate more meaningful results. In this paper, we use XML keys defined in [5] to improve the quality of diff algorithms. That is, XML keys determine which elements in different versions refer to the same entity in the real world, and therefore should be matched by the diff algorithm. We present an algorithm that extends an existing diff algorithm with a preprocessing phase for pairing elements based on keys.

1. Introduction

XML has become the standard format for data exchange on the Web. It helps the process of publishing data, especially when the underlying information is constantly being updated. The data consumer, on the other side, may be interested not only on the current contents of a web site, but also on the updates that have been made since his last access. As an example, one may want to know what are the new products in a catalog, or which products had their prices changed. To help the task of comparing two versions of XML documents, a number of diff algorithms have been proposed in the literature [1, 2, 15, 16].

The majority of these algorithms are based on a structural analysis of the documents. Similar to diff algorithms on strings, their main strategy is based on finding large fragments of data that are identical in both versions of a document and match them. After finding these matchings, a sequence of operations that transforms the old version of the document to the new one is generated. This is called an edit script or delta. In many applications, XML documents are not arbitrary tree structured data, but have well defined structure and semantics. A strategy based solely on structural and value similarities can generate erroneous matchings of elements.

We have conducted an experimental study to analyze the results of two diff algorithms: XyDiff[1], and X-Diff[15]. The experiments consisted of modifying an XML tree by inserting, deleting, modifying, and moving both internal and leaf nodes in the tree, and then analyzing how semantically meaningful the changes detected by these algorithms were. The results showed the following: 1) both algorithms are extremely sensitive to changes in the structure of the document, especially when they involve insertion and removal of internal nodes; 2) the existence of several similar or identical subtrees in a document may induce the algorithms to erroneously match them, and these erroneous matchings are propagated to their ancestors and descendants. We illustrate these problems below.

Example 1: Consider the two XML documents, represented as XML trees, depicted in Figure 1. They contain information about university professors. Each `professor` has a `name`, an `office`, and optionally a `phone` or `email`. Comparing the old version with the new one, we can observe that both `John` and `Mary` have their offices changed, and moreover, that `Mary` has moved to `John`'s former office. If the strategy of the diff algorithm is to find the largest common subtree in both versions, it will match `John`'s old office (node 9) with `Mary`'s new one (node 49). This matching can then be propagated upwards by matching `professor` elements (nodes 4 and 45). That is, the `professor` element node that corresponds to `John` is matched with the one that corresponds to `Mary`. As a consequence, the edit script contains an update on the `name` of the `professor` who works in the matched office, while our expected result is an update on `professor`'s office. In particular, if we know that `name` uniquely identifies a `professor` in the document, that is, `name` is a key for `professor`, then it is always the case that matches of `professor` elements based on their names produce more meaningful results than matches based on similar subtrees.

Observe that not only values have been modified, but the structure of the document has also changed. In the old version, professors are organized by their universities, while in the new one, they are placed under their departments. In our experimental study, the edit scripts generated by both XyDiff and X-Diff consisted of operations that remove all subtrees rooted at `professor` elements, followed by in-

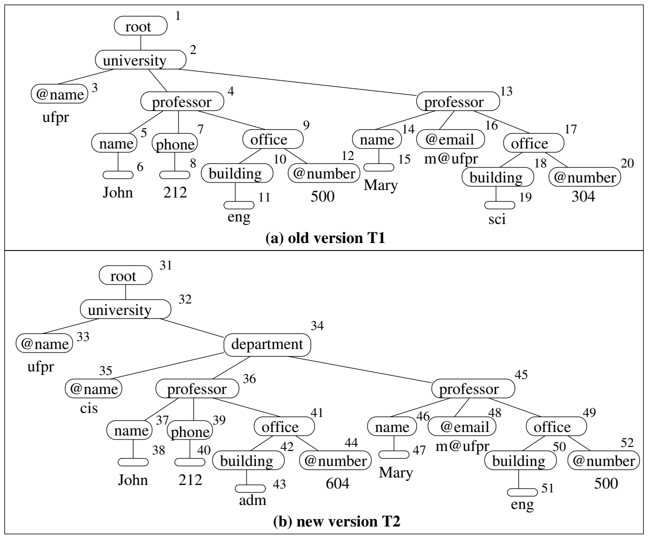


Figure 1. Two versions of XML trees

sertions of the same subtrees under the new `department` node. This is because in both algorithms only nodes reached by following exactly the same path from the root can be matched. Clearly, this is not the expected results from the semantical point of view, which is the creation of a new level in the tree. Similar to the previous case, if we know that `name` uniquely identifies a `professor` *no matter where the* `professor` *node occurs in the tree* then the correct matchings of `professor` elements would be found, avoiding their removals in the edit script. □

In this paper, we propose an algorithm, called XKeyMatch, that uses XML keys[5] to guide the comparison of XML documents. That is, first elements in the two versions are matched based on their key values, and then a structural analysis is performed to determine their differences. In Example 1 we would give as input to the diff algorithm that `name` uniquely identifies a `professor` in the entire document. A strategy based on keys is natural when comparing two relational databases. Since XML has become a standard for data exchange, it is natural to apply the same strategy for this format.

One area in which matchings based on semantics is especially important is data cleansing[11]. One of the main goals of data cleansing is to detect inconsistencies on input data. As an example, consider a datawarehouse that maintains data imported from external sources. Periodically these external sources update and publish new versions of their database. In order to keep the datawarehouse up-to-date, it is necessary to determine what are the differences of the new version compared to the previous one. If the previous version had been through data cleansing, one would like to know if previously detected mistakes have been corrected, either to prevent redoing the cleansing process, or to report the error to the external source. Ideally, a diff algorithm for helping this task should first identify which pieces in the two versions of imported data refer to the same entity and then determine what has been changed.

Contributions. The main contributions of this paper are:

- A proposal of a semantical approach in the context of diff algorithms for XML;
- An algorithm for matching elements in two versions of XML documents based on XML keys.

To the best of our knowledge this is the first algorithm that introduces keys in the context of XML diff algorithms, and that generates semantically meaningful results when there are changes in the structure of the document.

Related Work. A number of diff algorithms have been proposed in the literature both for text documents[8] and for tree structures[13, 14, 17]. XyDiff[1] is one of the earliest algorithms proposed for XML. It was designed for datawarehouses that store huge amounts of data, and therefore it was designed to be efficient, both in time and space. The algorithm is based on an ordered tree model. When the documents are accompanied with a DTD[4], attributes defined as identifiers (ID) are used to match elements according to their values. X-Diff[15] is an algorithm based on an unordered tree model. The distinguishing feature of diffX algorithm[2] is that it looks for matches in isolated fragments of the XML tree, instead of pairing nodes along a tree traversal. A different strategy for finding matches is applied by KF-Diff[16]. It is based on defining unique paths starting from the root for each node in the tree. Whenever such a path cannot be found, which happens when a node has more than one child with the same label, these labels are replaced by *key fields*. Key fields are values contained in the subtrees rooted at these nodes that can distinguish them among those with the same label. Although the idea of key fields is used in this algorithm, it is applied as a technique for deriving unique paths. They are not defined by the user, and therefore are not meant to capture the semantics of the document. Comparative studies of existing XML diff algorithms can be found in [7] and [12].

Organization. The rest of the paper is organized as follows. Section 2 defines XML keys, and presents definitions related to diff algorithms. Section 3 describes our proposal of a semantical diff algorithm, followed by our conclusions in Section 4.

2. XML Keys and Diff Algorithms

This section presents some definitions used throughout the paper.

Tree model and XML keys. XML documents can be modeled as trees. Nodes in the tree can be of three types: element, attribute, and text, where attribute labels are prefixed by "@". Based on node types, we define function $lab(n)$, and $val(n)$ as follows: if n is an element node then $lab(n)$

denotes the name of the element and $val(n)$ is undefined; if n is an attribute node then $lab(n)$ denotes the name of the attribute and $val(n)$ is its associated string value; if n is a text node then $lab(n)$ = "S", and $val(n)$ is its string value. Two examples of XML trees are illustrated in Figure 1.

To define a key we specify three things: 1) the *context* in which the key must hold; 2) a *target* set on which we are defining a key; and 3) the *values* which distinguish each element of the target set. For example, the key specification of Example 1 has a context of the `root` (the entire document), a target set of `professor`, and a single key value, `name`. Specifying the context node and target set involve path expressions.

The path language we adopt is a common fragment of regular expressions [10] and XPath [6]:

$$Q \quad ::= \quad \epsilon \quad | \quad l \quad | \quad Q/Q \quad | \quad //$$

where ϵ is the empty path, l is a node label, "/" denotes concatenation of two path expressions (*child* in XPath), and "//" means *descendant-or-self* in XPath. To avoid confusion we write $P//Q$ for the concatenation of P, $//$ and Q.

Following the syntax of [5] we write an XML key as:

$$K: \ (C, \ (T, \ \{P_1, \ldots, P_p\}))$$

where K is the name of the key, path expressions C and T are the context and target path expressions respectively, and $P_1, ..., P_p$ are the key paths. For the purposes of this paper, we restrict the key paths to be simple paths (without "//"). A key is said to be *absolute* if the context path C is the empty path ϵ, and *relative* otherwise.

Example 2: Using this syntax, some constraints on XML trees in Figure 1 can be written as follows:

- $k_1 \ : \ (\epsilon, (university, \{@name\}))$: within the context of the entire document (ϵ denotes the root), a `university` is identified by its `name`.
- $k_2 \ : \ (university, (//professor, \{name, phone\}))$: within the context of each subtree rooted at a `university` element, a `professor` is uniquely identified by the values of its subelements `name` and `phone`. The `professor` can appear anywhere in the subtree rooted at `university`.

□

To define the meaning of an XML key, we use the following notation: in an XML document (tree), $n[\![P]\!]$ denotes the set of node identifiers that can be reached by following path expression P from the node with identifier n. $[\![P]\!]$ is an abbreviation for $r[\![P]\!]$, where r is the root node of the tree. As an example, in Fig. 1(a), $[\![university]\!] = \{2\}$, $2[\![professor]\!] = \{4, 13\}$ and $[\![//name]\!] = \{5, 14\}$.

The formal definition of the meaning of an XML key is given next.

Definition 1 *An XML tree satisfies an XML key* (Q, (Q′, $\{P_1, \ldots, P_n\}$)) *iff for any n in $[\![Q]\!]$, and any n_1 and n_2 in $n[\![Q']\!]$, if for all* i, $1 \leq$ i $\leq n$ *there exists z_1 in $n_1[\![P_i]\!]$ and z_2 in $n_2[\![P_i]\!]$ such that $z_1 =_v z_2$, then $n_1 = n_2$.*

The definition above involves value equality on trees ($=_v$), so we formalize this notion below.

Definition 2 *Given an XML tree T, and two nodes n_1 and n_2 in T, we say that they are value equal, denoted as $n_1 =_v n_2$ iff the following conditions are satisfied: 1)* lab(n_1) = lab(n_2)*; 2) if n_1 and n_2 are attribute or text nodes then* val(n_1) = val(n_2)*; 3) if n_1 and n_2 are element nodes then: an attribute A is defined for n_1 iff it is also defined for n_2, and $val(n_1.A) = val(n_2.A)$; moreover, if* [$d_1, \ldots, d_k$] *are subelements of n_1 then n_2 has subelements* [$d'_1, \ldots, d'_k$]*, and for all $i \in [1, k]$ there exists $j \in [1, k]$ such that $d_i =_v d'_j$.*

For example, XML tree of Fig. 1(a) satisfies key $(//professor, \{name\})$ since $[\![//professor]\!] = \{4, 13\}$, and $4[\![name]\!] \neq_v 13[\![name]\!]$.

Diff Algorithms. The essential goal of a diff algorithm is to find an edit script such that given a version of a document in time $t-1$ and the edit script, it is possible to obtain its version in time t. The number of operations in a edit script is called the *edit distance*. Although the edit distance is often used to define the cost of the transformation, finding a minimum edit script is not always the best strategy for generating semantically meaningful results. In the next Section, we propose using XML keys to guide the node matching process of a diff algorithm.

3. A Semantical Diff Algorithm

In this section we present XKeyMatch, an algorithm for matching elements in two versions of XML trees based on XML keys. This algorithm is designed to be executed before a diff algorithm that compares the contents in both versions. The architecture of the system is depicted in Figure 2.

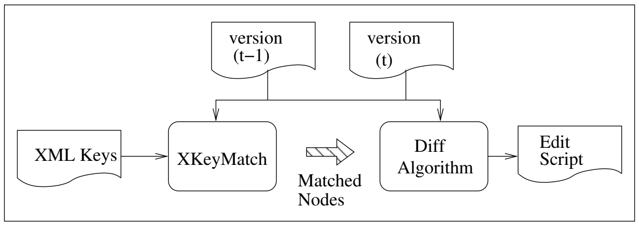


Figure 2. A diff algorithm with a preprocessing phase with XKeyMatch

Algorithm XKeyMatch receives as input two versions, v_{t-1} and v_t, of XML trees, and a set of XML keys Σ that are known to be satisfied by both versions. The output is a set Γ of node pairs $[n_1, n_2]$, where n_1 is a node in v_{t-1} that refers to the same entity as node n_2 in v_t according to Σ. The set Γ is then given as input to a diff algorithm that, based on these matchings, compares versions v_{t-1} and v_t and generates an edit script.

Algorithm XKeyMatch is based on the construction of a deterministic finite automaton (DFA) from the set Σ of XML keys, denoted as KeyDFA(Σ). Using this DFA, it is possible to process all keys in Σ at the same time, and all matchings based on Σ can be generated with a single preorder traversal on v_{t-1} and v_t. More specifically, each state of the DFA represents a set of paths, and it stores information on all keys that can be defined on nodes reached by following these paths. Therefore, each step of the XML tree traversal corresponds to a change of state in the automaton; information on keys stored at each state is used to collect nodes that are candidates for matchings based on these keys.

After collecting all candidates, algorithm XKeyMatch pairs nodes in both versions according to their key values. These steps of the algorithm are depicted in Figure 3. Given XML trees T_1 and T_2, and a set of XML keys Σ, the algorithm first builds KeyDFA(Σ) (Line 1). Then candidates are selected by calling function `get_candidates` for both trees (Lines 2 and 3). The resulting set of matches is computed by function `match`, which compares the candidates previously collected (Line 4). Next, we present each of these steps in detail.

```
Function XKeyMatch
Input: XML trees $T_1$, $T_2$, and a set of XML keys $\Sigma$
Output: a set of matched node pairs
1. KeyDFA:= DFA($\Sigma$);
2. candidates($T_1$):= get_candidates($T_1$, KeyDFA);
3. candidates($T_2$):= get_candidates($T_2$, KeyDFA);
4. return (match(candidates($T_1$), candidates($T_2$), $T_1$, $T_2$);
```

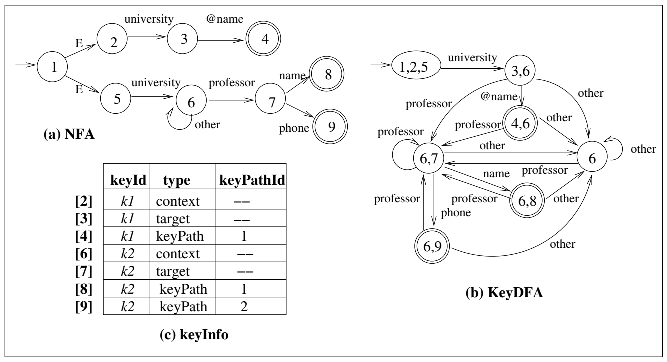
Figure 3. Algorithm XKeyMatch

DFA Construction. Given a set Σ of XML keys, this step of algorithm XKeyMatch generates a deterministic finite automaton, denoted as KeyDFA(Σ), where each state stores information for processing every key in Σ with a single traversal on an XML tree T.

Let $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$, where each σ_i is of the form $(Q_i, (Q'_i, \{P_i^1, \ldots, P_i^{n_i}\}))$. We first describe the construction of a non-deterministic finite state automaton (NFA) associated with each key σ_i in Σ. We start with the construction of a NFA for each path p in $\{Q_i, Q'_i, P_i^1, \ldots, P_i^{n_i}\}$, defined as $M(p) = (N_p, L_p \cup \{other\}, \delta_p, S_p, F_p)$, where N_p is a set of states, L_p is the alphabet, δ_p is the transition function, S_p is the start state, and F_p is the set of final states. Here, "*other*" is a special character that can match any character. These automaton have "linear structure"; that is, if $p = l_1/\ldots/l_m$ then $\delta_p(S_p, l_1) = q_1$, for each j, $1 \leq j < m$, $\delta_p(q_j, l_{j+1}) = q_{j+1}$, and $q_m = F_p$. If p contains "//" then there exists a transition from a state back to itself with "*other*". That is, if $p = \ldots//l_j\ldots$ for some j then $\delta_p(q_{j-1}, other) = q_{j-1}$, where $q_0 = S_p$. The final states of these NFAs carry information about the key considered for its construction, denoted as *keyInfo*. Let $F = \{F_{Q_i}, F_{Q'_i}, F_{P_i^1}, \ldots, F_{P_i^{n_i}}\}$. For each $f \in F$, $keyInfo[f]$ contains the following information:

- `keyId`: σ_i's identifier;
- `type`: the value of this field is *context* if $f = F_{Q_i}$, *target* if $f = F_{Q'_i}$ and *keyPath* otherwise;
- `keyPathId`: a identifier for each key path. This field is defined only when $keyInfo[f].type = keyPath$; in this case, if $f = F_{P_i^j}$ then $keyInfo[f].keyPathId = j$.

The NFA for key σ_i is obtained by making the final state of $M(Q_i)$ coincide with the start state of $M(Q'_i)$, and the final state of $M(Q'_i)$ coincide with start states of $M(P_i^k)$, $1 \leq k \leq n_i$. The NFA for all keys in Σ, $M(\Sigma)$ is finally obtained by creating a new start state with ϵ-transitions to the start states of all $M(\sigma_i)$, $1 \leq i \leq n$. An example of the resulting NFA for $\Sigma = \{k_1$ `:(university, {name})`, k_2 `:(university, (//professor, {name, phone}))`$\}$ is depicted in Figure 4(a), and the corresponding *keyInfo* structure is given in Figure 4(c).



	keyId	type	keyPathId
[2]	*k1*	context	––
[3]	*k1*	target	––
[4]	*k1*	keyPath	1
[6]	*k2*	context	––
[7]	*k2*	target	––
[8]	*k2*	keyPath	1
[9]	*k2*	keyPath	2

Figure 4. DFA construction

Given NFA $M(\Sigma)$, KeyDFA(Σ) is obtained applying standard subset construction algorithm[10]. The resulting automaton for our running example is shown in Figure 4(b). Observe that, although in $M(\Sigma)$ each state contains information of at most one key in Σ, after the conversion, each state q' in KeyDFA(Σ) contains information from all original states represented by q'. As an example, in Figure 4(b), $keyInfo(\{3, 6\}) = \{keyInfo[3], keyInfo[6]\}$.

The automaton construction described in this section is similar to that defined in [9] for XML stream processing, which evaluates the effectiveness of processing a large number of XPath expressions on streams when the DFA is constructed "lazily". Although the number of states in a DFA can grow exponentially on the number of input path expressions, the number of keys for a given document is usually small. Moreover, since changes on keys are not frequent, if versions of the same document are periodically downloaded, KeyDFA(Σ) can be locally stored, instead of being recomputed at each execution of the diff algorithm.

Selection of Candidates. Given KeyDFA(Σ), algorithm XKeyMatch process each of the XML trees given as input using this automaton, gathering information on possible candidates for matchings according to Σ. Let T be one these trees, and KeyDFA(Σ) = (Q, A, δ, q_0, F). Starting with the root of T and start state q_0, T is traversed in preorder, and each step in the tree traversal corresponds to a step in its processing with the automaton. During the traversal, information about key values are collected using data stored at each state q visited. That is, suppose the current state is q when processing a node n in T. If *keyInfo*$[q]$ contains information on a key path of a key k, then n is a key value for some node n_t, and therefore we can associate n_t with the value of n. Observe that k may contain more than one key path. If this is the case, then n_t is identified as a candidate for matching only if it is associated with values for all key paths of k, and we say that n_t is "keyed" on k.

To keep track of the information needed to determine whether a node is a candidate for matching along the tree traversal, we associate the following with each key $k = (Q, (Q', \{P_1, \ldots, P_n\}))$ in Σ, in a structure called $keyVal[k]$:

- contextNodes: a set of nodes in $[\![Q]\!]$;
- targetNode: last node visited in $n[\![Q']\!]$ for some n in contextNodes;
- keyNode: last node visited in targetNode$[\![P_i]\!]$.
- keyPathId: key path identifier i, $1 \leq i \leq n$;

Recall that a node in the tree may play different roles (as context, target, or key path) for different keys in Σ, and this information is given by *keyInfo*$[s]$, where s is a state in KeyDFA(Σ). Suppose that the current state of KeyDFA(Σ) is s when processing node n in T. Then for each element v in *keyInfo*$[s]$ we obtain the value of v.keyId $= k$, and values in v are used for filling up fields in $keyVal[k]$. As an example, consider tree T_1 in Figure 1(a). Let the current node in T_1 be 2 (a university node), and the current state of KeyDFA(Σ) be $\{3, 6\}$. Then the algorithm sets $keyVal[k_1]$.targetNode $= 2$, since *keyInfo*$[3]$ states that the current node is the target of k_1. Moreover, node 2 is inserted in $keyVal[k_2]$.contextNodes, based on the value of *keyInfo*$[6]$. That is, structure $keyVal[k]$ is a placeholder for information gathered along the tree traversal. Whenever values for all key paths of k are found, values in $keyVal[k]$ contain data on a candidate for matching based on k. The result of function get_candidates on T_1 and T_2 is given in Figure 5.

The first line in the table of Figure 5(a) has been collected according to the XML key k_1 : ((university, {@name})), while candidate 2 has been collected according to key k_2 : ((university, (//professor, {name, phone}))). Observe that node 13 (a professor node) has not been included in the set, since it does not have values for key path phone. Similarly, in Figure 5(b), line 1 has been collected using key k_1, while line 2 has been collected according to k_2.

keyId	contextNodes	targetNode	[keyNode, keyPathId]
k_1	1	2	{[3,1]}
k_2	2	4	{[5,1], [7,2]}

(a) get_candidates(T_1, keyDFA)

keyId	contextNodes	targetNode	[keyNode, keyPathId]
k_1	31	32	{[33,1]}
k_2	32	36	{[37,1], [39,2]}

(b) get_candidates(T_2, keyDFA)

Figure 5. Selection of candidates on T_1 and T_2

Matching. Given the set of candidates for matching from both versions of XML trees T_1 and T_2, function match looks for candidates in these sets with the same key values. That is, we search for nodes n_1 in candidates of T_1 and n_2 in candidates of T_2 that are keyed on the same key k and coincide on the values of all key paths. When k is a relative key, we can only conclude that n_1 matches n_2 if the contexts in which n_1 and n_2 are defined also match. Consider again the XML keys in our running example. If both candidates contain professor-nodes with the same name and phone, we can only conclude that they are indeed the same professor if the university (the context) in which they are defined also match. After finding a pair of target nodes $[n_1, n_2]$ that match, this matching is propagated downwards. That is, if they have been matched based on the values of some key paths, these key path nodes can also be matched.

In function match, all pairs of nodes that are candidates for matching are collected in a data structure containing the following information: the key identifier k (keyId); the context node in T_1 (contextNode$_1$); the target node in T_1 (targetNode$_1$); the context node in T_2 (contextNode$_2$); the target node in T_2 (targetNode$_2$); a set of records [keyPathId, keyNode$_1$, keyNode$_2$], where keyNode$_1$ and keyNode$_2$ are key nodes in T_1 and T_2, respectively, with the same value.

As an example, consider again XML trees T_1 and T_2 depicted in Figure 1 and the output of function get_candidates on T_1 and T_2 given in Figure 5. After comparing the values of candidates for matching, the resulting set contains the values shown in Figure 6. The first candidate is included in the set because nodes 2 and 32 are university nodes in T_1 and T_2, respectively, with the same @name value. Similarly, the second candidate is inserted because nodes 4 and 36 are professor nodes that coincide on the values of both name and phone. To compute the resulting set of matches, we start by inserting the pair $[1, 31]$, the roots of T_1 and T_2, in the set. When processing the first candidate $[2, 32]$ it is checked whether their context has already been matched. Since this is the case, $[2, 32]$ is inserted in the result. This matching is propagated downwards and pair $[3, 33]$ (@name nodes) are also inserted in the result. For the second candidate $[4, 36]$, it is checked whether their context $[2, 32]$ has already been matched. Since this is also the case, we can conclude the it is indeed a match and insert the pair in the resulting set. The propagation of matches includes in the result the following pairs: $[5, 37]$, $[6, 38]$ (name nodes), $[7, 39]$ and $[8, 40]$ (phone nodes).

keyId	context node$_1$	target node$_1$	context node$_2$	target node$_2$	[keyPathId, keyNode$_1$, keyNode$_2$]
1	1	2	31	32	{[1,3,33]}
2	2	4	32	36	{[1,5,37], [2,7,39]}

Figure 6. Candidates for matching

Implementation. Algorithm XKeyMatch has been implemented in C++, using DOM [3]. XyDiff is the algorithm chosen to take as input the set of matches resulting from XKeyMatch, find additional matchings, and generate the edit script. Some libraries from XyDiff have been used by XKeyMatch to implement the communication between them. We have conducted some experiments to evaluate the effectiveness of our proposal, and to determine if it indeed solves the problems detected in other diff algorithms and reported in Section 1. The results are very encouraging. In particular, for Example 1, the definition of a single key $(//professor, \{name\})$ prevents both problems described in the introduction. A final remark is that, although our proposal can have an impact on the performance of the diff algorithm to which algorithm XKeyMatch is applied to preprocess the input XML documents, many applications favor the quality of the result rather than the efficiency of the algorithm.

4. Conclusion

We have proposed a new approach in the context of diff algorithms for XML. As opposed to previous works, that are based solely on the structural analysis of XML documents, our technique takes into consideration their semantics. Our approach consists of extending the structural analysis with a preprocessing phase which uses XML keys to match elements that refer to the same entity in two versions of the document. Although XKeyMatch requires the user to be familiar with the documents being compared, when the input keys faithfully capture their semantics, our algorithm always generates more meaningful results than others based solely on structure and value similarities. One topic for future work is to perform an experimental study using large amounts of data, especially real ones, in order to determine the impact of the preprocessing phase in practice. Possible test beds are scientific databases, since they present appropriate structure and behavior.

References

[1] S. Abiteboul, G. Cobna, and A. Marian. Detecting changes in XML documents. *ICDE'02*, 2002.

[2] R. Al-Ekram, A. Adma, and O. Baysal. diffx: an algorithm to detect changes in multi-version xml documents. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 1–11. IBM Press, 2005.

[3] V. Apparao et al. Document Object Model (DOM) Level 1 Specification. W3C Recommendation, Oct. 1998.

[4] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation, Feb. 1998.

[5] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. *Computer Networks*, 39(5):473–487, Aug. 2002.

[6] J. Clark and S. DeRose. XML Path Language (XPath). W3C Working Draft, Nov. 1999.

[7] G. Cobena, T. Abdessalem, and Y. Hinnach. A comparative study for xml change detection, 2002.

[8] FSF. Gnu diff. Available at http://www.gnu.org/software/diffutils/diffutils.html.

[9] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[10] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[11] J. Maletic and A. Marcus. Data Cleansing: Beyond Integrity Analysis. *Proceedings of The Conference on Information Quality (IQ2000), Massachusetts Institute of Technology, Boston, MA, USA*, pages 200–209, 2000.

[12] L. Peters. Change detection in xml trees: a survey. Technical report, University of Twente, 2005.

[13] S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6:184–186, 1977.

[14] K. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 3(26):422–433, 1979.

[15] Y. Wang, D. J. DeWitt, and J. Cai. X-Diff: an effective change detection algorithm for XML documents. *ICDE*, pages 519–530, 2003.

[16] H. Xu, Q. Wu, H. Wang, G. Yang, and Y. Jia. Kf-diff+: Highly efficient change detection algorithm for xml documents. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 1273–1286, London, UK, 2002. Springer-Verlag.

[17] K. Zhang, R. Stgatman, and D. Shasha. Simple fast algorithm for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18:1245–1262, 1989.