Estimating XML version trees through maximum weight branching

曹 哲† 岩井原 瑞穂†

早稲田大学大学院情報生産システム研究科 〒808-0135 福岡県北九州市若松区ひびきの 2-7

E-mail: †asulahzala@gmail.com, †iwaihara@waseda.jp

ABSTRACT

XML is the de facto standard format for data exchange and manipulation of structured documents. Meanwhile, structured documents having past versions are rapidly growing, especially among the field of wiki contents and office documents. Even though there is an edit history for these user-generated contents, it is still hard to show how a document evolved by only comparing old versions with the latest version. To overcome this problem, we propose a version tree reconstruction mechanism scheme for XML documents. A version tree can explain how a document has evolved though collaborative editing, as well as illuminate dependencies among documents. This paper concentrates on reconstruction of XML version trees, and we also present an experiment on synthetic data to show how the reconstruction performs.

Keyword XML, maximum weight branching, similarity value, version tree

1. INTRODUCTION

Version is a description of an object during a period of time or under a certain point of view, whose recording is notable for the considered application. As the widespread diffusion of semi-structured data in XML format, structured documents having past versions are rapidly growing. Shared or interactive contents such as office documents and wiki contents are often provided with both the latest version and all past versions. Therefore, in order to check changed parts, old versions are compared with the latest version. Early works on XML documents mainly concentrate on similarity calculation for the purpose of grouping them into clusters. Deise et al. [1] proposed a version detection mechanism based on classification techniques. According to the similarity value between two files, their version detection mechanism seeks to verify whether these two files are versions of the same document. However, their work still does not cover the relationships of these versions, which means the proposed mechanism cannot clearly show the edit history of a document. In our proposal, this problem can be solved by the reconstruction of version trees.

A version tree is a directed tree in which each node represents one version, and its structure reflects edit history of the document. The structure of a version tree is shown in Fig. 1. Here, it should be noted that a version history sometimes becomes a DAG (directed acyclic graph) rather than a tree. This occurs when two versions are merged into a new version. In this paper, we don't consider this merging structure. By reconstructing XML version trees, we can explain how a document evolved by collaborative editing. In this process, firstly, we compute similarity values between each two versions using defined similarity measures. Secondly, we build a directed graph in which each node represents one version and each the edge weight represents the similarity value between two versions. Thirdly, we simplify the directed graph by removing arcs having small weights. At last we can obtain an XML version tree by using the maximum weight branching [3].

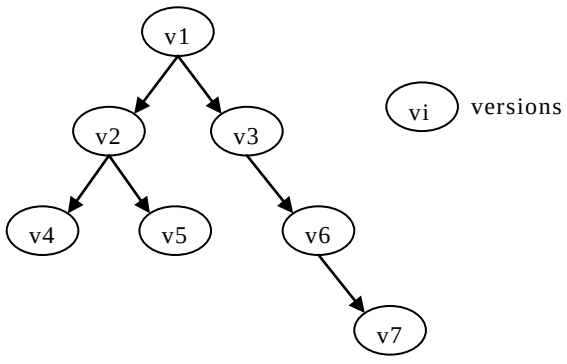


Fig.1 XML version tree

We can evaluate the accuracy of the proposed version tree reconstruction mechanism scheme by comparing the generated estimated version tree with the true version tree.

The main contributions of this paper are:

- We show a method for reconstructing a version tree about a document with multiple versions. By the version tree we can see how a document has evolved clearly rather than compare two versions frequently.
- We utilize the maximum weight branching to compute the version tree.

The paper is organized as follows. In Section 2, we will define the main issues about the problem. Section 3 describes the version tree reconstruction algorithm. Our experiment result is shown in Section 4. Finally, concluding remarks are discussed in Section 5.

2. PROBLEM DEFINITION

In this section, before discussing how to reconstruct version trees in detail, we first give several definitions regarding our problem.

Definition 1. (*XML Document*). We model an *XML document* as a rooted ordered tree where nodes are labeled as either element, text or attribute, and only element nodes can have a child node.

One element consists of element name, attribute list and a text value. In this paper we consider that the content consists both attribute and value of an element.

In this paper, an XML document may have various versions and all versions are managed as separate files.

Definition 2. (*Matched, Changed, Removed, Added and Totalnum*). We define two elements as *matched* if both their structure and content are the same, *changed* if their structure remains the same while their content changes. An element is *removed* if it is deleted from one version, an elements is *added* if it is a new element. We define that *Totalnum* is the sum of *matched, changed, removed* and *added* elements.

Definition 3. (*Similarity Value*).We define the *similarity value* between two versions V_1 and V_2 as:

$$SIM(V_1, V_2) = w_1 \cdot M + w_2 \cdot C + w_3 \cdot R + w_4 \cdot A$$

C=*changed elements/Totalnum*

M=*matched elements/Totalnum*

R=*removed elements/Totalnum*

A= *added elements/Totalnum*.

Here, M is the percentage of matched elements, C is the percentage of changed elements, R is the percentage of removed elements and A is the percentage of added elements, the sum M, C, R and A is 1. Therefore, we use $M = 1 - C - R - A$ to reduce variables.

We can also easily see from the function that the interval of variables M, C, R and A is from 0 to 1. Analyzing the ranges of M, R and A, we can obtain the fact that $SIM(V_1, V_2)$ ranges from $min[w_3, w_4]$ to w_1. In order to uniformly distribute these similarity values, we can map these values to values $\in [0,1]$ by using the normalized formula which is defined as follows.

Definition 4. (*normalized similarity value*). We define *normalized similarity value* between two versions as:

$$SIM(V_1, V_2) = w_1 \cdot (1 - C - R - A) + w_2 \cdot C + w_3 \cdot R + w_4 \cdot A$$
$$Normalize\ SIM = (SIM - min[w_3, w_4])/(w_1 - min[w_3, w_4])$$

Later, we will use this normalized formula to calculate the similarity value in experiments in order to set these values from 0 to 1. The function in Definition 3 is similar to the idea presented in [1], which uses content similarity value and the percentage of added and deleted elements to calculate structure similarity value. Below, we give a simple example to show how the function works.

```
<Actors>                          <Actors>
  <Actor>                           <Actor>
    <Name>                            <Name>
      <FirstName>Mike</FirstName>       <FirstName></FirstName>
      <LastName>Johnson</LastName>      <LastName>Johnson</LastName>
    </Name>                           </Name>
    <Movies>                          <Movies>
      <Title>movie1</Title>             <Title>movie3</Title>
      <Title>movie2</Title>             <Title>movie2</Title>
    </Movies>                         </Movies>
  </Actor>                          </Actor>
</Actors>                         </Actors>
        Version#1                         Version#2
```

Fig.2 Two sample versions

Fig.2 shows two versions of a document, where the left part is version1 and the right part is version2. First, we use the XML diff tool of [2] to detect the number of matched, changed, added and removed elements so as to compute M, C, A and R. The diff result is shown in Fig.3.

```
<xd:xmldiff version="1.0">
  <xd:node match="1">
    <xd:node match="1">
      <xd:node match="1">
        <xd:node match="1">
          <xd:remove match="1"/>
        </xd:node>
      </xd:node>
      <xd:node match="2">
        <xd:node match="1">
          <xd:change match="1">movie3</xd:change>
        </xd:node>
      </xd:node>
    </xd:node>
  </xd:node>
</xd:xmldiff>
```

Fig.3 Diff result of Fig.2

From Fig 3, we can see that <name>movie1</name> is changed elements, while <FirstName>Mike<FirstName> is changed elements. According to Definition 3, the total number of elements is 8, *C and R* equals to 1/8=0.125, *A* equals 0, $1 - C - R - A = 0.75$. If we set the weight factors w_1, w_2, w_3, w_4 as 0.8, -0.2, -0.2, -0.2, the similarity value between two versions $SIM(V_1, V_2)$ is 0.55, the normalized similarity value is 0.75.

Definition 5. (*Version Graph*).A *version graph*, denoted as V_G, is a directed weighted graph $G = (V, E, w)$, where: *V* is the set of all versions, *E* is the set of version edges and $w(e)$ is the a function that gives a similarity value to every edge $e \in E$.

Definition 6. (*Maximum Weight Branching*). Given a directed graph *G*, a *branching B* is a subgraph of *G* such that *B* is a directed forest. The *maximum weight branching* is a *branching* of maximum weight in *G*

Definition 7. (*Version Tree*). A *version tree*, denoted as T_V, is a directed, rooted tree in which all edges originate from the root *R* where *R* refers to the initial version of a document.

Basically, there are three types of version tree estimation:

1. **With timestamps.** This situation mainly occurs in wiki contents where all the timestamps are available for each version of a document. The tree structure is embedded in the linear structure of the timestamp ordering.
2. **Without timestamps**. This situation occurs when dealing with documents that lost their timestamps, due to actions like copying.
3. **Timestamps partially available**. This situation occurs when a subset of versions retain their timestamps. Surviving timestamps can be a constraint on version trees.

In this paper, we mainly discuss version tree estimation without timestamps.

3. PROPOSED METHOD

Our proposed method for reconstructing XML version trees performs three processes. The first process is *similarity calculation*, which is responsible for building the version graph. The second process is called *graph simplification* and it mainly deals with deleting redundant edges by setting an appropriate threshold. Finally, maximum weight branching is applied to construct the version tree during the *algorithm application* process. The whole process is depicted in Fig.4.

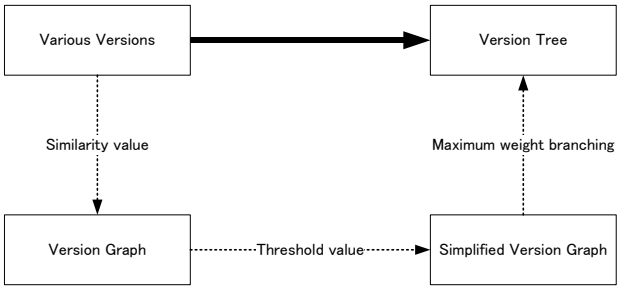


Fig.4 Process of proposed method

3.1 Similarity Calculation

In this paper, we use similarity values to indicate the similarity between two versions both in structure and content. The function of similarity value has been given in Definition 4. The factors (w_1, w_2, w_3, w_4) are defined based on the importance of the four features in different situations.

The weights for C, R and A (w_2, w_3, w_4) are negative values because the smaller these values the more similar versions. Also, it should be noted that *M, C, R, A* are not uniquely determined and their values vary depending on the diff algorithm.

3.2 Graph Simplification

Because a version graph is a complete graph, decreasing the number of edges is necessary to reduce computation cost.

In this phase, we remove edges which have small weights. A threshold is given to achieve the purpose of graph simplification. The edges with weight higher than the threshold remain, while the lower ones are removed. The threshold will be empirically determined.

3.3 Algorithm Application

Here, we first give several definitions in order to explain how to find the maximum weight branching.

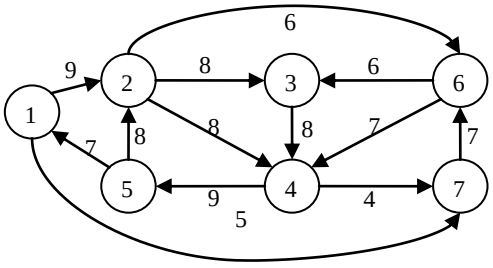
Definition 8. (*Critical Arc*). Given a directed graph $G=(V,E)$, suppose that e is an arc from i to j, the source of e $s(e)$ is i and the terminal of e $t(e)$ is j. Then e is a *critical arc* if the weight of e is not less than the weight of any other arc whose terminal is also vertex j.

Definition 9. (*Critical Graph*). Given a directed graph G, a subgraph is a *critical graph* of G if the subgraph consists of the set of all *critical arcs* chosen in G.

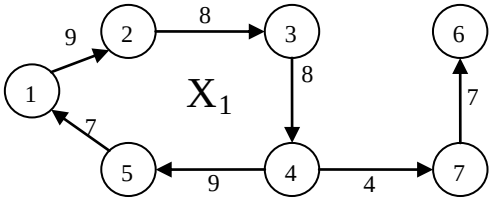
Let H be a critical graph in G with weight function w, and let the cycles in H be $C_i(i=1,2,\ldots,k)$. Let W be the set of vertices in G that do not belong to any of the cycles in H. Replace each cycle C_i in H by a single vertex X_i. Let $V_1=\{X_1,X_2,\ldots,X_k\}\cup W$. If e is an arc in G that is not an arc of C_i and $t(e)$ is a vertex of C_i, define $w_1(e)=w(e)-w(f)+w(e_i)$, where f is the unique arc in Ci that $t(e)=t(f)$ and e_i is an arc of minimum weight among all arcs in that cycle. If $t(e)$ is not a vertex of these k cycles, $w_1(e)=w(e)$.

Definition 10. (*Condensed Graph*). Given a directed graph G, the *condensed graph* of G is the weighted multigraph G_1 that is constructed with V_1 as the set of vertices with the revised weight function w_1.

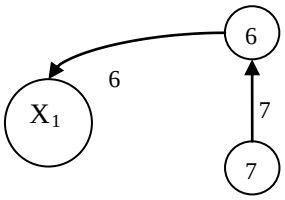
Here, we will show a small example to explain these definitions in Fig. 5. The numbers represent seven versions of a document.



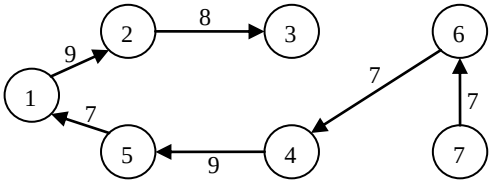
(a) Version Graph



(b) Critical Graph



(c) Condensed Graph



(d) Maximum Weight Branching

Fig. 5 (a)(b)(c)(d) Maximum branching computation

Basically, the maximum weight branching algorithm can be divided into two phases: condensation phase and unraveling phase. In the condensation phase, the input is the weighted digraph $G=G_0$. Construct G_1 from G_{i-1} by condensing the cycles in its critical digraph. G_k is the first digraph in the sequence for which the critical graph H_k is acyclic. In the unraveling phase, the graph H_k is a maximum weight branching in G_k. Let $B_k=H_k$. Construct B_{i-1} from B_i by expanding the condensed cycles. B_i is a maximum weight branching in G_i for $i=k,k-1,k-2,\ldots,1,0$. The output is $B=B_0$.

Definition 11. (*Arborescence*). A branching $B=(V,E)$ is an *arborescence* if there is exactly one vertex (the root of the arborescence) with indegree equal to zero.

We can see from Definition 10, a spanning arborescence is nothing other than a rooted tree. A digraph is called quasi-strongly connected if for every pair of vertices u and v in the digraph there exists a vertex w such that there are directed paths from w to u and from w to v. It has been shown that a digraph has an arborescence if and only if it is quasi-strongly connected. In this situation, we can use Edmonds's Algorithm to construct the version tree. Tarjan described an efficient implementation of Edmonds's algorithm in [4]. The algorithm can be implemented to run in time $O(mlog\ n)$, where n is the number of vertices of the graph and m is the number of edges. With a slight modification, the implementation can be made to run in time $O(n^2)$, which is preferable when dealing with dense graphs. In [5], Gabow et al. give an $O(nlog\ n+m)$ time implementation of Edmonds's algorithm for finding an optimum spanning arborescence. We also note that a better time complexity cannot be achieved by any

implementation of Edmonds's algorithm since Edmonds's algorithm can be used to sort n numbers (sorting n numbers by comparison requires $\Omega(n\log n)$ time, and since we always have to look at every edge of the graph, we cannot achieve a better time complexity for Edmonds's algorithm than $O(n\log n + m)$.

4. EXPERIMENTS AND EVALUATION

In our experiments we use synthetic data and the experiments are carried out as follows:

(1) Data Acquisition

This phase is responsible for acquiring different versions of a document. In order to obtain these versions, for one document d, we generate a random sequence Q of updating operations (insert, delete and modify). Then a *true version tree* can be obtained by applying Q to d. in our experiment, we generate six versions for one document. The structure of the *true version tree* used in the experiment is shown in Fig.6.

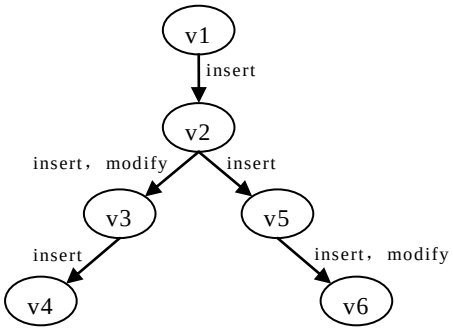


Fig.6 the true version tree of our experiment

(2) Data Processing

In this phase, we apply the three processes presented in Fig.4 to deal with these different versions. First, we use Microsoft XML diff tool to detect the number of matched, changed, added and removed elements so as to compute C, A and R. Then we apply the normalized similarity calculation formula to obtain the similarity values between each two versions. Table.1 shows the normalized similarity values between each two versions of one document with different weight factors. Here, symmetric matrix will obtained if we set w_3 equals w_4 which is shown in Table.1 (a).

The next step is to check whether there are similarity values that are far smaller than average value. We delete these edges for the purpose of simplifying the version graph. Based on the version graph, we construct various version trees using Edmonds's algorithm. Fig.7 shows all the estimated version trees generated from Table.1.

After constructing estimated trees, we compare them with the true version tree to calculate the precision and recall. In this paper, we use ancestor-descendant relationships to do precision and recall calculation.

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	—	0.91	0.84	0.78	0.84	0.78
V_2	0.91	—	0.92	0.85	0.92	0.85
V_3	0.84	0.92	—	0.93	0.85	0.79
V_4	0.78	0.85	0.93	—	0.79	0.74
V_5	0.84	0.92	0.85	0.79	—	0.93
V_6	0.78	0.85	0.79	0.74	0.93	—

(a) (w_1, w_2, w_3, w_4) equals (0.8, -0.2, -0.2, -0.2)

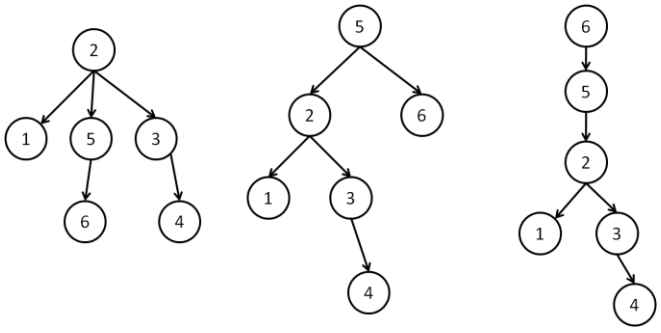
	V_1	V_2	V_3	V_4	V_5	V_6
V_1	—	0.93	0.87	0.82	0.87	0.82
V_2	0.91	—	0.94	0.88	0.94	0.88
V_3	0.84	0.92	—	0.94	0.87	0.82
V_4	0.78	0.85	0.93	—	0.81	0.77
V_5	0.84	0.92	0.87	0.82	—	0.94
V_6	0.78	0.85	0.81	0.77	0.93	—

(b) (w_1, w_2, w_3, w_4) equals (0.8, -0.2, -0.2, 0)

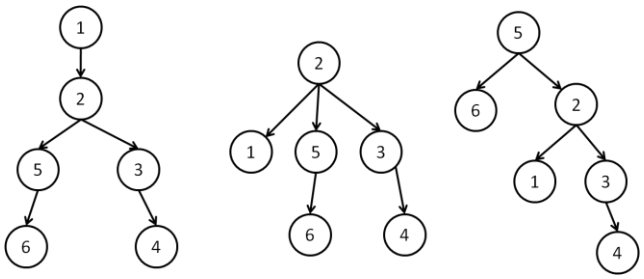
	V_1	V_2	V_3	V_4	V_5	V_6
V_1	—	0.91	0.84	0.78	0.84	0.78
V_2	0.93	—	0.92	0.85	0.92	0.85
V_3	0.87	0.94	—	0.93	0.87	0.81
V_4	0.82	0.88	0.94	—	0.82	0.77
V_5	0.87	0.94	0.87	0.81	—	0.93
V_6	0.82	0.88	0.82	0.77	0.94	—

(c) (w_1, w_2, w_3, w_4) equals (0.8, -0.2, 0, -0.2)

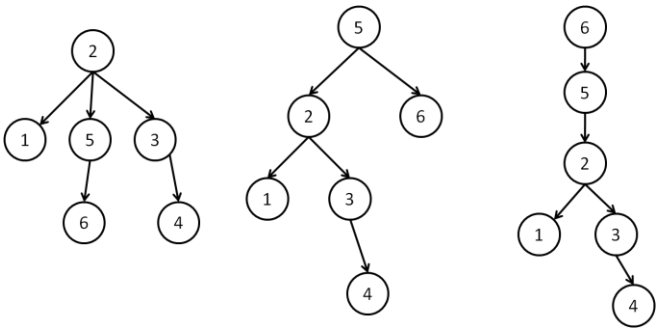
Tab.1 (a)(b)(c) Similarity values with different weight factors



(a) (w_1, w_2, w_3, w_4) equals (0.8, -0.2, -0.2, -0.2)



(b) (w_1, w_2, w_3, w_4) equals (0.8, -0.2, -0.2, 0)



(c) (w_1, w_2, w_3, w_4) equals (0.8, -0.2, 0,0.2)

Fig.7 (a)(b)(c) Version trees with different weight factors

Definition 12. (*Precision*). Given an estimated version tree $T_v = (V, E_v)$ and a true version tree $T_0 = (V, E_0)$, let E_v^*, E_0^* be the set of all descendant-ancestor relationships, define $Precision = |E_v^* \cap E_0^*| / |E_v^*|$.

Definition 13. (*Recall*). Given an estimated version tree $T_v = (V, E_v)$ and a true version tree $T_0 = (V, E_0)$, let E_v^*, E_0^* be the set of all descendant-ancestor relationships, define $Recall = |E_v^* \cap E_0^*| / |E_0^*|$.

Using Definition 12 and 13, we list all the accuracy and recall values of the estimated version trees in Table. 2.

(3) Data Analysis

From Tab.2, we can conclude that with weight factors (0.8, -0.2, -0.2, 0), better accuracy is achieved (we even obtain the true version tree with root node 1).Here, setting w_3 to 0 means that we give more weight on the insert operation and in fact, we generate the six versions mainly by insert operations. Through the above analysis, we can see that setting appropriate weight factors is quite important to obtain better estimated version trees. In our future work, SVM can be applied to determine these weight factors.

Fig.5	(a)			(b)			(c)		
Root node	2	5	6	1	2	5	2	5	6
Precision	0.9	0.4	0.2	1	0.9	0.4	0.9	0.4	0.2
Recall	0.5	0.4	0.3	1	0.5	0.4	0.5	0.4	0.3

Tab.2 Accuracy of estimated version trees

It is also easy to note that we cannot necessarily obtain spanning tree structures in a directed graph with a fixed root node, which indicates that in some situations we can only obtain distributed branching rather than a tree structure.

5. CONCLUSION

This paper focuses on a version tree reconstruction method for XML documents. The importance of the problem is increasing under the background that structured documents having past versions are rapidly growing. In this paper, we define a similarity calculation function in order to detect the similarity of two versions. Moreover, version trees with different root nodes are also constructed so as to show how a document has evolved.

As future work, we are going to apply the version tree reconstruction method to Wikipedia contents and develop a weight factor selection system in order to obtain estimated version trees with higher accuracy. Also, we will try to optimize the similarity function and the maximum weight branching algorithm in order to make the method applicable in different domains.

6. REFERENCES

[1] Deise de Brum Saccol, Nina Edelweiss, Renata de Matos Galante and Carlo Zaniolo, “XML Version Detection”, DocEng’07, August 28–31, 2007, Winnipeg, Manitoba, Canada.

[2] http://msdn.microsoft.com/en-us/library/aa302294.aspx

[3] J.Edmonds, “Optimum branching”, Journal of Research of the National Bureau of Standards, 71B, 1967, p.233-240

[4] R. E. Tarjan, “Finding optimum branching”, Networks, 7(1):25-35, 1977.

[5] H.N. Gabow, Z. Galil, T. Spencer, and R.E. Tarjan, “Efficient algorithms for _finding minimum spanning trees in undirected and directed graphs”, Combinatorica, 6(2):109-122, 1986.