

# XDiff

Ulf Wehling

## Was versteht man unter XDiff ?

Unter XDiff versteht man einen Algorithmus der in der Lage ist Veränderungen zwischen zwei XML Dokumenten zu ermitteln. Zur Erkennung von Veränderungen zwischen XML Dokumenten gibt es verschiedene Algorithmen, die sich in zwei Kategorien einteilen lassen. Zum einen die Algorithmen die das geordnete Baum Modell (*ordered tree model*) benutzen und zum anderen die Algorithmen, die das ungeordnete Baum Modell (*unordered tree model*) benutzen.

In geordneten Bäumen ist neben der Vater-Kind Beziehung zwischen Knoten auch die Anordnung der Kindknoten von Bedeutung. In ungeordneten Bäumen ist nur die Vater-Kind Beziehung von Bedeutung, die Anordnungsreihenfolge der Kindknoten ist unerheblich.

## Motivation durch ein Beispiel

In den nächsten Jahren wird XML HTML als Standardsprache im Internet ablösen. Da sich viele online Informationen regelmäßig ändern, wird ein Tool benötigt, daß in der Lage ist große Massen an XML Dokumenten effizient auf Veränderungen gegenüber früheren Dokumentversionen zu untersuchen.

Um das Problem deutlicher zu machen betrachten wir folgendes Beispiel:

Angenommen ein Vater ist daran interessiert Bücher für seine Kinder auf einer online Auktionsseite zu ersteigern. Beim ersten Besuch auf der Seite erhält er eine Liste der angebotenen Bücher inklusive verschiedener Informationen zu diesen Büchern, wie z.B. der aktuelle Bietpreis usw.. Zwei Stunden später erhält er eine modifizierte Version der online Seite und benutzt nun das Tool um die Veränderungen zwischen beiden Dokumentversion darzustellen.

<pre>&lt;Buecher&gt; &lt;Buch&gt;   &lt;Title&gt;Harry Potter und der Stein der Weisen&lt;/Title&gt;   &lt;Autor&gt;J.K.Rowling&lt;Autor&gt;   &lt;Verkaeuffer&gt;     &lt;Name&gt;Mike&lt;/Name&gt;   &lt;/Verkaeuffer&gt;   &lt;akt_Gebot Restzeit="36 Std."&gt;8.50€&lt;/akt_Gebot&gt;   &lt;Bieter&gt;     &lt;Name&gt;Stefan&lt;/Name&gt;   &lt;/Bieter&gt; &lt;/Buch&gt; &lt;Buch&gt;   &lt;Title&gt;Die Abenteuer von Tom Sawyer&lt;/Title&gt;   &lt;Autor&gt;Mark Twain&lt;/Autor&gt;   &lt;Verkaeuffer&gt;     &lt;Name&gt;Christian&lt;/Name&gt;   &lt;/Verkaeuffer&gt;   &lt;akt_Gebot Restzeit="4 Std."&gt;3.50€&lt;/akt_Gebot&gt;   &lt;Bieter&gt;     &lt;Name&gt;Tim&lt;/Name&gt;   &lt;/Bieter&gt; &lt;/Buch&gt; &lt;/Buecher&gt;</pre>	<pre>&lt;Buecher&gt; &lt;Buch&gt;   &lt;Title&gt;Die Abenteuer von Tom Sawyer&lt;/Title&gt;   &lt;Autor&gt;Mark Twain&lt;Autor&gt;   &lt;Verkaeuffer&gt;     &lt;Name&gt;Christian&lt;/Name&gt;   &lt;/Verkaeuffer&gt;   &lt;akt_Gebot Restzeit="2 Std."&gt;4.50€&lt;/akt_Gebot&gt;   &lt;Bieter&gt;     &lt;Name&gt;Tim&lt;/Name&gt;   &lt;/Bieter&gt; &lt;/Buch&gt; &lt;Buch&gt;   &lt;Title&gt;Harry Potter und der Stein der Weisen&lt;/Title&gt;   &lt;Autor&gt;J.K.Rowling&lt;/Autor&gt;   &lt;Verkaeuffer&gt;     &lt;Name&gt;Mike&lt;/Name&gt;   &lt;/Verkaeuffer&gt;   &lt;akt_Gebot Restzeit="34 Std."&gt;10.00€&lt;/akt_Gebot&gt;   &lt;Bieter&gt;     &lt;Name&gt;Mark&lt;/Name&gt;   &lt;/Bieter&gt; &lt;/Buch&gt; &lt;/Buecher&gt;</pre>
---	--

Abb. 1.1: Alte Dokumentversion

Abb. 1.2: Modifizierte Dokumentversion

Zuerst prüft das Tool ob die beiden Dokumentversionen identisch sind. Ist dies nicht der Fall, so wird jedes „Buch“-Segment der alten Version mit jedem der neuen Version verglichen, um herauszufinden welche Bücher noch verfügbar sind, wie hoch der aktuelle Bietpreis ist usw.

In obigen Beispiel sind beide Bücher noch verfügbar, auch wenn sich ihre Reihenfolge im Dokument verändert hat. Als nächstes wird für jedes noch vorhandene Buch geprüft, welche Informationen sich geändert haben. Im Falle des Beispiels sollte der Benutzer – in obigem Fall der Vater – darüber informiert werden, daß sich die verbleibende Zeit zum Ersteigern der Bücher um 2 Stunden verringert hat und, daß der Bietpreis für das Harry Potter Buch aktuell bei 10€ liegt und sich der Preis um 1,5€ gegenüber der ersten Dokumentversion erhöht hat.

## Überblick

Ein solches Tool kann für ein Anfragesystem auf zweierlei Weisen nützlich sein:

- Zum einen wenn ein Benutzer eine dauerhafte Anfrage an eine sich von Zeit zu Zeit ändernde Datenquelle hat.  
In diesem Fall kann das Tool zur Berechnung eines *delta* (Menge von Veränderungen zwischen 2 Dokumenten) benutzt werden. Da die Größe des *delta* in der Regel kleiner ist als die Größe des Originaldokuments kann die Auswertung der Anfrage schneller durchgeführt werden.
- Zum anderen ist man oft nur an einer speziellen Änderung in Dokumenten interessiert. In diesem Fall kann das Tool zum herausfiltern der unwichtigen Daten benutzt werden und die für den Benutzer wichtigen Daten schnell darstellen.

Im folgenden betrachten wir den X-Diff Algorithmus von Wang, DeWitt und Cai etwas näher. Die folgenden Merkmale sind Schlüsselmerkmale von X-Diff:

- Der strukturierte Aufbau von XML Dokumenten, d.h. die Tatsache, daß sich jedes XML Dokument als Baum darstellen läßt.  
Mithilfe der Knotensignatur und eines *matching* ist der Algorithmus in der Lage ein kostenminimales *matching* zu finden und daraus ein minimales Bearbeitungsskript (*edit script*) zu generieren mit dessen Hilfe man die Originalversion des Dokuments in die neue überführen kann.
- Da sich XML Dokumente als Bäume darstellen lassen, läßt sich die Erkennung von Änderungen zwischen zwei XML Dokumenten auf die Unterschiede zwischen den zwei Baumdarstellungen zurückführen (*tree-to-tree correction problem*). Algorithmen, die Unterschiede zwischen zwei Bäumen berechnen können, lassen sich – wie schon erwähnt – in zwei Kategorien einteilen, je nachdem ob sie auf geordneten oder ungeordneten Bäumen arbeiten.  
Für Datenbankapplikationen ist das ungeordnete Baum Modell aber von größerer Bedeutung als das geordnete. Gleiches gilt für viele XML Dokumente.  
Der X-Diff Algorithmus ist in erster Linie dazu entwickelt worden um ungeordnete Bäume auf Unterschiede hin zu überprüfen. Das ist der Hauptunterschied zu anderen Arbeiten auf diesem Gebiet. Die meisten anderen Algorithmen zur Erkennung von Veränderungen zwischen XML Dokumenten arbeiten nur auf geordneten Bäumen korrekt.

- Das Erkennen von Veränderungen zwischen ungeordneten Bäumen ist erheblich schwerer als das Erkennen von Veränderungen zwischen geordneten Bäumen. Zhang und andere konnten 1992 nachweisen, daß das Erkennen von Veränderungen auf ungeordneten Bäumen im Normalfall ein NP-vollständiges Problem ist. Durch die Ausnutzung einiger spezieller Eigenschaften von XML Dokumenten - die sie von normalen beschrifteten ungeordneten Bäumen unterscheiden – schafft X-Diff es aber in polynomieller Laufzeit das optimale *delta* zwischen zwei Dokumenten zu berechnen.

## Andere Algorithmen zu diesem Thema

Bevor wir zu X-Diff kommen betrachten wir noch kurz zwei weitere Algorithmen zu diesem Thema, die aber beide für geordnete Bäume konzipiert worden sind. XMLTreeDiff berechnet ebenfalls die Unterschiede zwischen zwei XML Dokumenten. Zuerst wird jedem Knoten der beiden Dokumente mithilfe der Funktion DOMHash ein Hashwert zugewiesen, der als Signatur der Knoten dient. Als nächstes werden die beiden Bäume verkleinert, indem identische Unterbäume (z.B. Unterbäume mit gleichem Hashwert), zwischen denen keine Änderung in beiden Dokumenten besteht, gelöscht. Danach wird der Algorithmus von Zhang und Shasha benutzt um die Unterschiede zwischen beiden Bäumen zu finden. Der Algorithmus von Zhang und Shasha erwartet als Eingabe zwei geordnete Bäume und berechnet das minimale *delta* zwischen beiden Bäumen in Zeit  $O(|T_1|*|T_2|*\min\{\text{depth}(T_1),\text{leaves}(T_1)\}*\min\{\text{depth}(T_2),\text{leaves}(T_2)\})$ . Der von diesen beiden entwickelte Algorithmus ist zur Zeit der effizienteste Algorithmus im Bezug auf die Erkennung von Veränderungen zwischen zwei geordneten Bäumen. Aufgrund der Tatsache, daß in der ersten Phase des XMLTreeDiff Algorithmus identische Teilbäume aus den beiden Bäumen entfernt werden kann es zu Konflikten, auf die wir hier nicht näher eingehen, mit dem Algorithmus von Zhang und Shasha kommen. Daraus folgt, daß XMLTreeDiff nicht immer das optimale bzw. ein korrektes *delta* berechnet.

Cobéna und andere haben den Algorithmus XyDiff entwickelt. Dieser Algorithmus berechnet zunächst für jeden Knoten durch ein bottom-up Vorgehen eine Signatur (Hashwert) und ein Gewicht (die Größe des jeweiligen Unterbaums). Dann startet der Algorithmus mit den beiden Wurzelknoten und vergleicht die Signaturen der beiden Knoten auf Gleichheit. Sind sie identisch so werden sie *matched*. Andernfalls kommen die Kindknoten in eine priority queue. In dieser priority queue werden die Unterbäume mit dem höchsten Gewicht immer zuerst auf Gleichheit geprüft. Immer wenn zwei identische Unterbäume gefunden worden sind versucht der Algorithmus das *matching* auf die Vaterknoten zu erweitern. Gibt es mehr als einen passenden Kandidatenknoten so sucht sich der Algorithmus – aufgrund einiger einfachen Heuristiken – einen der passenden Kandidaten heraus. Der Algorithmus arbeitet sehr effizient und hat eine Laufzeit von  $O(n*\log n)$ , wobei n die Anzahl der Knoten im zweiten Dokument ist. Der Algorithmus erzielt auf geordneten Bäumen durchweg gute Ergebnisse. Aufgrund der einfachen Auswahlregeln die der Algorithmus benutzt kommt es aber oft zu falschen Auswertungen, das heißt auch XyDiff garantiert kein optimales Ergebnis.

Beide Algorithmen sind darauf spezialisiert Veränderungen zwischen geordneten Bäumen zu erkennen. Weiterhin finden beide Algorithmen oft auch nicht das optimale *delta*, d.h. die Menge von Operationen, die es erlaubt die Originalversion des Dokuments in die neue Version zu überführen.

Bevor wir uns nun die genaue Arbeitsweise des X-Diff Algorithmus von Wang, DeWitt und Cai anschauen müssen wir noch ein paar Dinge definieren.

## Baumdarstellung von XML Dokumenten

Basierend auf der Spezifikation des *Document Object Model* (DOM) lassen sich XML Dokumente durch Bäume repräsentieren. Man unterscheidet dabei drei verschiedene Knotentypen:

- 1) Elementknoten:  
Elementknoten werden durch Ovale dargestellt.  
Elementknoten sind keine Blattknoten. Sie sind durch einen Namen gekennzeichnet.
- 2) Textknoten:  
Textknoten werden durch Rechtecke mit durchgezogenen Linien dargestellt.  
Hierbei handelt es sich um Blattknoten mit einem Wert.
- 3) Attributknoten:  
Attributknoten werden durch Rechtecke mit gestrichelten Linien dargestellt.  
Hierbei handelt es sich ebenfalls um Blattknoten, die sowohl einen Namen als auch einen Wert haben.

Laut DOM Spezifikation sind Text- und Elementknoten geordnet, Attributknoten hingegen ungeordnet.

Die meisten Algorithmen für das Baum zu Baum Korrekturproblem (*tree-to-tree correction problem*) können nicht auf ungeordnete Bäume angewendet werden, da die Korrektheit der Ergebnisse von der Anordnung der Kindknoten innerhalb eines Dokuments abhängt.

Betrachte hier auch die beiden Abbildungen 1.1 und 1.2. In diesem Beispiel ist die Reihenfolge der Bücher vertauscht worden, was hier aber irrelevant ist.

Zwei Bäume werden als isomorph bezeichnet, wenn sie bis auf die Anordnungsreihenfolge der Kindknoten identisch sind.

## Operationen auf DOM Bäumen

Man definiert drei Basisoperationen auf DOM Bäumen:

- 1)  $\text{Insert}(x(\text{name}, \text{value}), y)$   
- fügt Knoten  $x$  mit Name  $\text{name}$  und Wert  $\text{value}$  als Kind von  $y$  ein
- 2)  $\text{Delete}(x)$   
- löscht Knoten  $x$
- 3)  $\text{Update}(x, \text{new\_value})$   
- verändert den Wert des Knotens  $x$  zu  $\text{new\_value}$

Zu beachten ist hier, daß alle Basisoperationen nur auf Blattknoten definiert sind. Zur Vereinfachung führt man auch noch die folgenden beiden Operationen ein, die sich aber auch mithilfe der drei Basisoperationen darstellen lassen:

- 1)  $\text{Insert}(T_x, y)$   
- Teilbaum  $T_x$  mit Wurzel  $x$  wird unter Knoten  $y$  eingefügt
- 2)  $\text{Delete}(T_x)$   
- löscht Teilbaum  $T_x$  mit Wurzel  $x$

Die Definition der 3 Basisoperationen stimmt im wesentlichen mit der Definition der Operationen überein, auf deren Basis XyDiff bzw. XMLTreeDiff arbeiten, mit Ausnahme der move-Operation. Diese Operation wird hier nicht benötigt, da die beiden erwähnten Algorithmen sie in erster Linie dazu verwenden Unterbäume von

einer Position an eine andere zu verschieben. Da X-Diff auf dem ungeordneten Modell basiert wird diese Operation hier nicht benötigt.

### Bearbeitungsskript

Unter einem Bearbeitungsskript versteht man eine Folge von Basisoperationen, die einen Baum in einen anderen überführen.

Beispiel für ein Bearbeitungsskript:

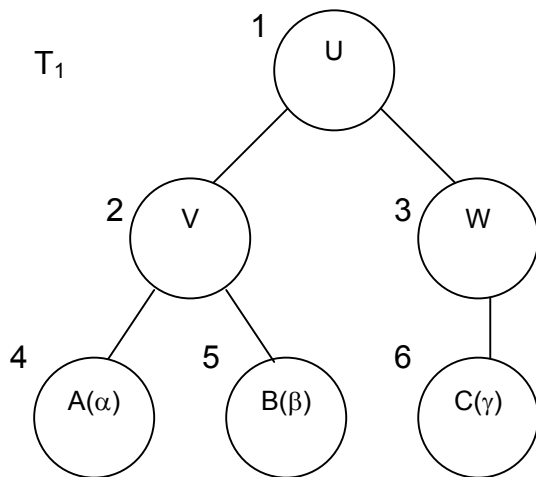


Abb. 2.1

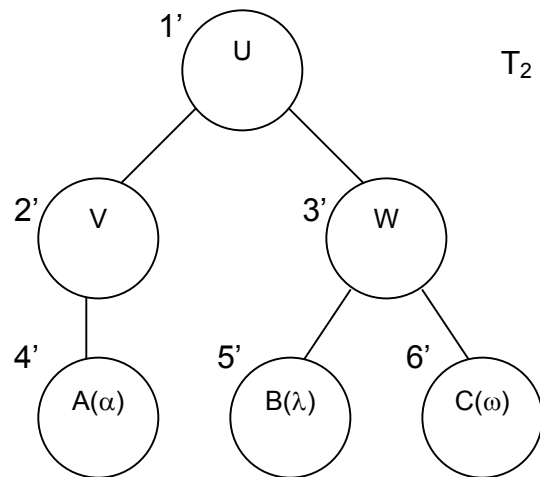


Abb. 2.2

Betrachte die obigen beiden Bäume  $T_1$  und  $T_2$ . Die Großbuchstaben sind dabei die Namen der Knoten und die griechischen Buchstaben stehen für den Wert des Knotens. Das folgende Bearbeitungsskript transformiert  $T_1$  zu  $T_2$ :

$$E(T_1 \rightarrow T_2) = \text{Delete}(5), \text{Insert}(5(B, \lambda), 3), \text{Update}(6, \omega)$$

Ein weiteres Bearbeitungsskript das den Baum  $T_1$  in den Baum  $T_2$  überführt, ist z.B.

$$E'(T_1 \rightarrow T_2) = \text{Update}(5, \lambda), \text{Delete}(5), \text{Insert}(5(B, \lambda), 3), \text{Update}(6, \omega)$$

### Generelles Kostenmodell für Bearbeitungsskripte

Wie aus dem obigen Beispiel ersichtlich wird, kann es viele verschiedene Bearbeitungsskripte für ein und dieselben Bäume geben. Das Ziel besteht nun darin, daß Skript mit den minimalen Kosten zu finden. Hierzu benutzt man folgende Definition für ein Kostenmodell:

Die Kosten eines Bearbeitungsskriptes  $E$  hängen von der Anzahl der im Skript vorkommenden Basisoperationen ab. Der Einfachheit halber wird angenommen, daß alle Operationen die gleichen Kosten haben. Die Kosten von  $E$  werden mit  $\text{cost}(E) = n$  bezeichnet, falls  $E = O_1 \dots O_n$  ist und  $O_i$  Basisoperationen sind.

Unter einem kostenminimalen Bearbeitungsskript bzw. optimalen Bearbeitungsskript versteht man nun das Berechnungsskript, daß den Baum  $T_1$  in den Baum  $T_2$  überführt und gemäß der obigen Definition die geringsten Kosten hat.

Für obiges Beispiel ist  $E$  das kostenminimale Bearbeitungsskript.

Weiterhin definiert man die Bearbeitungsdistanz (*editing distance*) zwischen zwei Bäumen wie folgt:

$\text{Dist}(T_1, T_2) = \text{cost}(E)$  wobei  $E$  das kostenminimale Bearbeitungsskript für  $T_1 \rightarrow T_2$  ist.

Beide Definitionen können auch auf Unterbaumpaare erweitert werden. Sei  $E$  ein Bearbeitungsskript, das den Teilbaum  $T_x$  in den Teilbaum  $T_y$  überführt.  $E$  wird kostenminimales Bearbeitungsskript genannt, wenn für alle Bearbeitungsskripte  $E'$  die  $T_x$  nach  $T_y$  überführen  $\text{cost}(E') \geq \text{cost}(E)$  ist. Die Bearbeitungsdistanz ist dann  $\text{Dist}(T_x, T_y) = \text{Cost}(E)$ .

### Knotensignatur und kostenminimales *matching*

Um die Veränderungen zwischen zwei Bäumen zu finden müssen wir zuerst versuchen zusammengehörige Knoten (Knoten die übereinstimmen) in beiden Bäumen zu finden. Bei der Suche nach übereinstimmenden Knoten wird nicht willkürlich vorgegangen. Aus dem Kontext in dem ein Knoten steht, können schon gewisse Schlüsse gezogen werden, so braucht man z.B. (man stelle sich die Abb. 1.1 und 1.2 als Bäume vor) keinen „Titel“-Knoten mit einem „Autor“-Knoten zu vergleichen, da man aufgrund ihrer Stellung innerhalb des Baums schon weiß, dass diese Knoten nicht übereinstimmen können. Desweiteren braucht man auch keine unterschiedlichen Knotentypen auf Übereinstimmung zu prüfen.

Beachtet werden sollte aber, dass es nicht ausreicht nur den Knotentyp und den Knotennamen zu vergleichen und die Knoten – sofern Typ und Name gleich sind – als gleich anzusehen und sie deshalb zu *matchen*. Die Beziehung des jeweiligen Knoten zu seinem Vaterknoten muß auch betrachtet werden.

Als Beispiel betrachten wir nun wieder die Abb 1.1 und 1.2, wobei wir uns die Dokumente wieder als Baum vorstellen. Dort sollte z.B. nicht ein Knoten „Name“ unterhalb eines „Verkaeuffer“ Knotens mit einem Knoten „Name“ unterhalb eines „Bieter“ Knotens als *gematched* werden, auch wenn die Werte übereinstimmen. Der „Name“ Knoten unterhalb des „Verkaeuffer“ Knotens hat eine andere Bedeutung als der „Name“ Knoten unterhalb eines „Bieter“ Knotens.

Deshalb wird hier die Knotensignatur definiert um zwei Knoten auf Übereinstimmung zu testen. Die Signatur eines Knotens  $x$  setzt sich dabei aus den Namen der Knoten von der Wurzel des Baums bis zum Knoten  $x$  und dem Knotentyp von  $x$  zusammen.

Formal bedeutet dies:

Sei  $x$  ein Element- oder Attributknoten in einem DOM Baum  $T$ , so gilt:

$$\text{signature}(x) = \text{Name}(x_1) \cdot \text{Name}(x_2) \cdot \dots \cdot \text{Name}(x_n) \cdot \text{Name}(x) \cdot \text{Type}(x)$$

wobei  $x_1$  die Wurzel von  $T$  und  $(x_1, \dots, x_n, x)$  der Pfad von der Wurzel zu  $x$  ist und  $\text{Type}(x)$  den Knotentyp von  $x$  bezeichnet.

Sei  $x$  ein Textknoten in einem DOM Baum, dann gilt:

$$\text{signature}(x) = \text{Name}(x_1) \cdot \text{Name}(x_2) \cdot \dots \cdot \text{Name}(x_n) \cdot \text{Type}(x)$$

Die Signatur für einen Knoten erhält man nun, indem man die obigen Werte konkateniert. Da alle Vorgängerknoten eines Blattknotens selbst keine Blattknoten sein können, braucht der Typ dieser Knoten auch nicht betrachtet zu werden, da es sich um Elementknoten handeln muß.

X-Diff sieht zwei Knoten als gleich an, wenn ihre Signaturen übereinstimmen.

Betrachte hierzu die Abbildung 2.1. Dort hat z.B. der Knoten 4 die Signatur „U.V.A.\$Attribute\$“ und der Knoten 3 die Signatur „U.W.\$Element\$“.

Als nächstes definieren wir den Begriff des *matching*.

Eine Menge  $M$  von Knotenpaaren  $(x, y)$  wird *matching* von  $T_1$  nach  $T_2$  genannt, falls gilt:

- 1)  $\forall (x, y) \in M, x \in T_1, y \in T_2, \text{signature}(x) = \text{signature}(y)$
- 2)  $\forall (x_1, y_1) \in M \text{ und } (x_2, y_2) \in M \text{ gilt: } x_1 = x_2 \leftrightarrow y_1 = y_2$
- 3)  $\forall (x, y) \in M \text{ und } x' \text{ ist Vaterknoten von } x \text{ und } y' \text{ ist Vaterknoten von } y \text{ gilt:}$

$$(x', y') \in M$$

Aus den obigen Kriterien folgt, daß Kindknoten nur dann *matchen*, wenn ihre Vaterknoten *matchen*. Die Kriterien 1) und 3) reduzieren den Platzverbrauch des *matching* und machen den Algorithmus effizient.

Basierend auf einem *matching* M kann ein Bearbeitungsskript für  $T_1$  nach  $T_2$  erzeugt werden. Knoten aus  $T_1$ , die nicht in M enthalten sind, werden gelöscht (delete). In  $T_2$  neu eingefügte Knoten werden eingefügt (insert) und Knotenpaare aus M deren Wert unterschiedlich ist werden aktualisiert (update).

Beispiel:

Die folgende Abbildung zeigt das *matching*  $M = \{(1, 1'), (2, 2'), (3, 3'), (5, 4'), (7, 7')\}$  zwischen zwei Bäumen  $T_1$  und  $T_2$ .

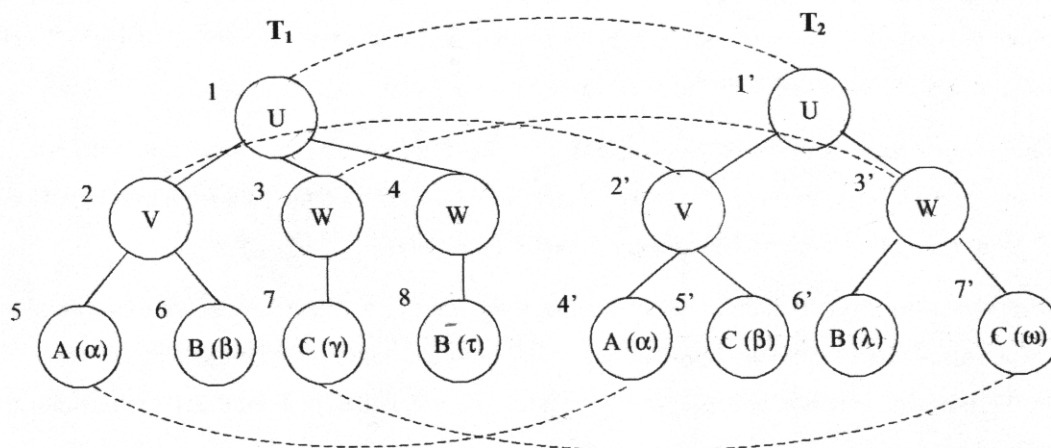


Abb. 3: Beispiel für *matching*

$(6, 5')$  kann z.B. nicht im M sein, weil die Knoten 6 und  $5'$  unterschiedliche Signaturen haben ( $\text{signature}(6) = \text{"U.V.B.\$Attribute\$"}$ ; und  $\text{signature}(5') = \text{"U.V.C.\$Attribute\$"}$ ). Die Knoten 8 und  $6'$  haben zwar die gleiche Signatur,  $(8, 6')$  existiert aber nicht in M, da das Knotenpaar  $(4, 3')$  (sind die Vaterknoten von 8 und  $6'$ ) nicht in M existieren.

Nun läßt sich zeigen, daß die Bearbeitungsdistanz zwischen zwei Blattknoten 0 ist, wenn ihre Signaturen und ihre Werte übereinstimmen. Stimmen die Signaturen überein, die Werte aber nicht, so beträgt die Differenz 1.

Damit läßt sich nun auch beweisen (das tun wir hier aber nicht), daß es ein *matching* gibt, aus dem wir ein kostenminimales Bearbeitungsskript erzeugen können.

Das zur Abb. 3 gehörige Bearbeitungsskript E sieht wie folgt aus:

$E = \text{Delete}(6), \text{Delete}(8), \text{Delete}(4), \text{Insert}(5'(C, \beta), 2), \text{Insert}(6'(B, \lambda), 3), \text{Update}(7, \omega)$

### Arbeitsweise von X-Diff

Seien im folgenden  $\text{DOC}_1$  und  $\text{DOC}_2$  XML Dokumente und  $T_1$  bzw.  $T_2$  die zu den Dokumenten gehörigen Bäume.

Zuerst prüft X-Diff, ob  $\text{DOC}_2$  verschieden ist von  $\text{DOC}_1$ . Sind die Dokumente verschieden, so findet X-Diff ein kostenminimales *matching* und generiert daraus ein kostenminimales Bearbeitungsskript für  $T_1 \rightarrow T_2$ .

Die Vorgehensweise von X-Diff läßt sich in 4 Phasen unterteilen:

- 1) Zuerst werden beide XML Dokumente geparkt und die zugehörigen Baumrepräsentationen werden erstellt. Während des Parsens wird für jeden Knoten ein Hashwert ermittelt, der zur Repräsentation des gesamten Unterbaums des

aktuellen Knoten benutzt wird.

- 2) In dieser Phase werden die XHashwerte der Wurzelknoten verglichen. Stimmen diese überein, so sind beide Dokumente identisch. Sind sie nicht identisch, so reduziert X-Diff die Größe der beiden Bäume  $T_1$  und  $T_2$ , indem er identische Unterbäume des zweiten Levels (auf Level 1 der beiden Bäume befinden sich nur die Wurzelknoten) rausfiltert.
- 3) X-Diff findet ein kostenminimales *matching*  $M_{\min}(T_1, T_2)$  zwischen beiden Bäumen.
- 3) In der letzten Phase wird basierend auf  $M_{\min}(T_1, T_2)$  aus Phase 2 ein kostenminimales Bearbeitungsskript  $E$  für  $T_1 \rightarrow T_2$  erstellt.

```
Input: (DOC1, DOC2)
/* Parsing and Hashing */
1. Parse DOC1 to T1 and hash T1;
   Parse DOC2 to T2 and hash T2;
/* Checking and Filtering */
2. a) If (XHash(Root(T1))=XHash(Root(T2)))
     DOC1 and DOC2 are equivalent, stop.
     Else go to b).
     b) For any second-level node pair(x1, x2)
         If (XHash(x1)=XHash(x2))
             Remove subtree rooted at x1 from T1;
             Remove subtree rooted at x2 from T2;
/* Matching */
3. Alg. Matching - Find a minimum-cost matching  $M_{\min}(T_1, T_2)$  from T1 to T2.
/* Generating minimum-cost edit script */
4. Alg. EditScript - Generate the minimum-cost edit script E from
    $M_{\min}(T_1, T_2)$ .
```

### X-Diff Algorithmus im Überblick

Nun betrachten wir die einzelnen Phasen etwas genauer.

### Parsing und Hashing

In dieser Phase werden  $DOC_1$  und  $DOC_2$  in zwei DOM Bäume  $T_1$  und  $T_2$  überführt. DOM Bäume enthalten viele unnötige Informationen und verbrauchen auch mehr Speicherplatz als andere Baumdarstellungen. Da sie aber eine bessere Schnittstelle zur Anbindung an Applikationen bieten werden sie hier trotz der obigen Nachteile verwendet.

Während des Parsens benutzt X-Diff die spezielle Hashfunktion XHash zum berechnen eines Hashwertes für jeden Knoten. Genau wie bei der Benutzung von DOMHash repräsentiert der XHashwert den gesamten Unterbaum des aktuellen Knoten. Der Unterschied zu DOMHash besteht aber in der Tatsache, daß zwei isomorphe Bäume den gleichen XHashwert bekommen, was bei der Verwendung von DOMHash nicht der Fall sein muß.

Im folgenden nehmen wir an, daß  $\text{hash}(\text{string})$  eine normale Hashfunktion ist, die einen string als Eingabe erwartet und einen string fester Länge als Hashwert ausgibt. Sei  $x$  ein Blattknoten, entweder ein Textknoten oder ein Attributknoten. Dann wenden wir die Hashfunktion auf die Konkatenation des Knotentyps mit seinem Namen und seinem Wert an:

$$\text{XHash}(x) = \text{hash}(\text{Type}(x).\text{Name}(x).\text{Value}(x))$$

Für einen Elementknoten  $x$ , der eine Liste von Kindknoten  $x_1, \dots, x_n$  hat, erhält man den XHashwert wie folgt:

- 1) Zuerst berechnet man  $\text{XHash}(x_1), \dots, \text{XHash}(x_n)$ .

- 2) Dann sortiert man die XHashwerte, so daß  $XHash(x_{1'}) \leq \dots \leq XHash(x_{n'})$  ist, wobei  $x_{1'}, \dots, x_{n'}$  die Knoten nach der Sortierung sind.
- 3) Konkateniere nun  $Type(x)$  und  $Name(x)$  mit den sortierten XHashwerten und berechne so den XHashwert für  $x$ :  
 $XHash(x) = hash(Type(x).Name(x).XHash(x_{1'}) \dots XHash(x_{n'}))$ .

Durch diese Art der Berechnung der XHashwerte für jeden Knoten kann jeder XHashwert eines Elementknotens seinen kompletten Unterbaum repräsentieren.

## Prüfen und Filtern

Zwei Bäume (Unterbäume) deren XHashwert übereinstimmt können nun als identisch angesehen werden. Hier werden nun die XHashwerte der beiden Wurzelknoten verglichen um zu prüfen ob die beiden Dokumente identisch sind. Stimmen beide Werte überein, so sind die beiden Bäume entweder äquivalent oder identisch. Stimmen beide Werte nicht überein, so muß es Unterschiede zwischen beiden Bäumen (beiden Dokumenten) geben.

Nun benutzt X-Diff die XHashwerte der Knoten des zweiten Levels um identische Unterbäume rauszufiltern und so die Größe der Bäume zu verkleinern. Beachtet werden muß hier aber, daß nur Unterbäume, die direkt unter der Wurzel hängen rausgefiltert werden dürfen. Aufgrund des dritten *matching* Kriteriums folgt, daß Unterbäume in tieferen Levels, trotz gleicher XHashwerte, nicht im *matching* existieren müssen und deshalb dürfen diese nicht rausgefiltert werden.

Beispiel:

Das folgende Beispiel zeigt die beiden obersten Level zweier XML Dokumente als DOM Bäume. Die gestrichelten Linien symbolisieren die Knoten mit den gleichen Hashwerten. Die zweite Abbildung zeigt, wie die Bäume aussehen, nachdem die äquivalenten Unterbäume rausgefiltert worden sind.

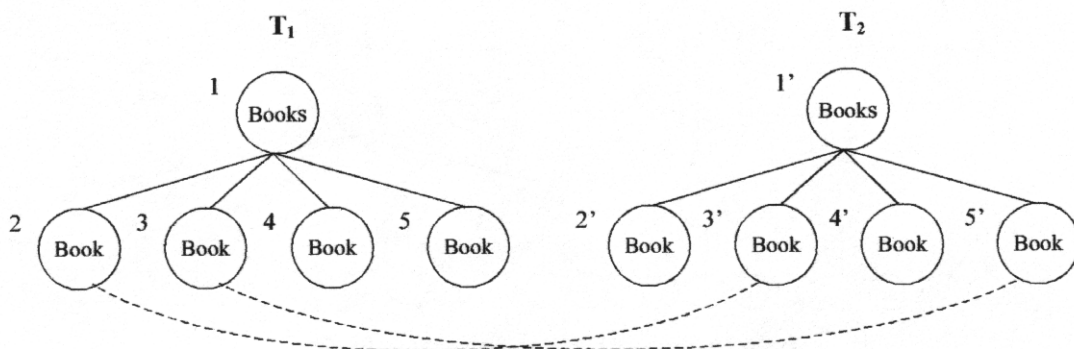


Abbildung 4.1: äquivalente Teilbäume des zweiten Levels



Abbildung 4.2: Bäume, nach dem rausfiltern

## Matching

In dieser Phase wird der *matching* Algorithmus benutzt um ein kostenminimales *matching* zwischen  $T_1$  und  $T_2$  zu finden.

Zuerst wird die Bearbeitungsdistanz für jedes Blattknotenpaar berechnet.

Als nächstes wird die Bearbeitungsdistanz für jedes Unterbaumpaar berechnet. So erhalten wir das kostenminimale *matching* zwischen den beiden verglichenen Unterbäumen.

Zum Schluß berechnet der Algorithmus die Bearbeitungsdistanz zwischen  $T_1$  und  $T_2$  und erhalten so das kostenminimale *matching*  $M_{\min}(T_1, T_2)$ .

Die Bearbeitungsdistanz zwischen zwei Blättern bzw. zwei Unterbäumen wird in einer Entfernungstabelle (Distance Table) gespeichert. Zur Berechnung der Bearbeitungsdistanz wird der „minimum-cost-maximum-flow“ Algorithmus benutzt. Zu beachten ist weiterhin, daß nur die Bearbeitungsdistanz zwischen zwei Knoten mit identischer Signatur berechnet werden muß. Aus diesem Grund erreicht der Algorithmus polynomielle Laufzeit. Müßte die Bearbeitungsdistanz für alle Knotenpaare berechnet werden, so hätte der Algorithmus keine polynomielle Laufzeit mehr.

```
Input: Tree  $T_1$  and  $T_2$ .
Output: a minimum-cost matching  $M_{\min}(T_1, T_2)$ .
Initialize: set initial working sets
   $N_1 = \{ \text{all leaf nodes in } T_1 \}$ ,  $N_2 = \{ \text{all leaf nodes in } T_2 \}$ .
  Set the Distance Table  $DT = \{ \}$ .
/* Step 1: compute editing distance for  $(T_1 \rightarrow T_2)$  */
Do {
  For every node  $x$  in  $N_1$ 
    For every node  $y$  in  $N_2$ 
      If (Signature( $x$ )=Signature( $y$ ))
        Compute Dist( $x, y$ );
        Save matching ( $x, y$ ) with Dist( $x, y$ ) in DT.
  Set  $N_1 = \{ \text{parent nodes of previous nodes in } N_1 \}$ ;
  Set  $N_2 = \{ \text{parent nodes of previous nodes in } N_2 \}$ .
} While (both  $N_1$  and  $N_2$  are not empty)
/* Step 2: mark matchings on  $T_1$  and  $T_2$ . */
Set  $M_{\min}(T_1, T_2) = \{ \}$ 
If (Signature(Root( $T_1$ )) $\neq$ Signature(Root( $T_2$ )))
  Return; /*  $M_{\min}(T_1, T_2) = \{ \}$  */
Else
  Add(Root( $T_1$ ), Root( $T_2$ )) to  $M_{\min}(T_1, T_2)$ .
  For every non-leaf node mapping  $(x, y) \in M_{\min}(T_1, T_2)$ 
    Retrieve matchings between their child nodes that are stored in DT.
    Add child node matchings into  $M_{\min}(T_1, T_2)$ .
```

## Matching Algorithmus

### Generierung eines kostenminimalen Bearbeitungsskriptes

Basierend auf dem in der vorherigen Phase gefundenen kostenminimalen *matching* wird nun ein minimales Bearbeitungsskript erstellt. Die Generierung erfolgt rekursiv, wobei bei den Wurzelknoten begonnen wird.

```
Input: Tree  $T_1$  and  $T_2$ , a minimum-cost matching  $M_{\min}(T_1, T_2)$ ,
      the distance table DT.
Output: an edit script E.
```

```

Initialize: set E=NULL;
x=Root(T1), y=Root(T2).
If ((x,y) ∉ Mmin(T1,T2))
  Return "Delete(T1), Insert(T2)". /* Subtree deletion and insertion */
Else If (Dist(T1,T2)=0)
  Return "";
Else {
  For every node pair (xi,yj) ∈ Mmin(T1,T2),
  xi is a child node of x, yj is a child node of y.
  If (xi and yj are leaf nodes)
    If (Dist(xi,yj)=0)
      Return "";
    Else
      Add Update(xi,Value(yj)) to E; /* Update leaf node */
  Else
    Add EditScript(Txi,Tyj) to E; /* Call subtree matching */
  Return E;
  For every node xi not in Mmin(T1,T2)
    Add "Delete(Txi)" to E;
  For every node yj not in Mmin(T1,T2)
    Add "Insert(Tyj)" to E;
  Return E.
}

```

## EditScript Algorithmus

### Laufzeitanalyse

Seien im folgenden  $|T_1|$  und  $|T_2|$  die Anzahl der Knoten in  $T_1$  bzw.  $T_2$ .

Für die Laufzeit der einzelnen Phasen gilt:

#### 1. Parsing und Hashing:

Die Zeit zum parsen und erstellen der Bäume beträgt  $O(|T_1| + |T_2|)$ . Bevor man die XHashwerte der Elementknoten bestimmen kann, muß man die XHashwerte der Kindknoten sortieren. Daher ergibt sich als Gesamtlaufzeit für diese Phase  $O(|T_1| \cdot \log(|T_1|) + |T_2| \cdot \log(|T_2|))$ .

#### 2. Prüfen und Filtern:

In dieser Phase vergleicht der Algorithmus die XHashwerte der beiden Wurzelknoten und filtert äquivalente Teilbäume des zweiten Levels in der Zeit  $O(|T_2|)$  heraus.

#### 3. Matching:

In dieser Phase wird die Bearbeitungsdistanz zwischen jedem Knotenpaar  $(x,y)$  (mit  $x \in T_1, y \in T_2, \text{signature}(x) = \text{signature}(y)$ ) bei den Blattknoten beginnend berechnet. Das kostenminimale matching erhält man, wenn man die Bearbeitungsdistanz zwischen den beiden Wurzelknoten gefunden hat.

Die Kosten zur Berechnung der Bearbeitungsdistanz zwischen zwei Blattknoten ist  $O(1)$ . Die Kosten zur Berechnung der Bearbeitungsdistanz zwischen allen Blattknotenpaaren sind somit  $O(|T_1| \cdot |T_2|)$ . (1)

Die Bearbeitungsdistanz zwischen zwei Elementknoten  $(x,y)$  (mit  $x \in T_1, y \in T_2, \text{signature}(x) = \text{signature}(y)$ ) erhält man, indem man ein kostenminimales matching zwischen den Kindknoten findet. Sei nun  $\text{deg}(x)$  der Ausgangsgrad des Knoten  $x$ . Die Kosten zur Berechnung der Bearbeitungsdistanz zwischen  $x$  und  $y$  betragen  $O(\text{deg}(x) \cdot \text{deg}(y) \cdot \max\{\text{deg}(x), \text{deg}(y)\} \cdot \log(\max\{\text{deg}(x), \text{deg}(y)\}))$

Angenommen es gibt  $M$  gleiche Elementknotensignaturen zwischen  $T_1$  und  $T_2$ .

Diese bezeichnet man mit  $S_1$  bis  $S_M$ .  $N_{1k}$  und  $N_{2k}$  bezeichnen die Anzahl der Knoten in  $T_1$  und  $T_2$ , die die Signatur  $S_k$  haben.  $x_{ki}$  und  $y_{kj}$  sind die Knoten mit der Signatur

$S_k$ . Die Kosten zur Berechnung der Bearbeitungsdistanz zwischen allen Elementknotenpaaren betragen

$$\sum_{k=1}^M \sum_{i=1}^{N_{1k}} \sum_{j=1}^{N_{2k}} O(\deg(x_{ki}) * \deg(y_{kj}) * \max\{\deg(x_{ki}), \deg(y_{kj})\} * \log(\max\{\deg(x_{ki}), \deg(y_{kj})\})) \quad (3)$$

$\deg(T_1)$  und  $\deg(T_2)$  bezeichnen nun den maximalen Ausgangsgrad in  $T_1$  und  $T_2$ . Dann gilt:

$$(3) \leq \sum_{k=1}^M \sum_{i=1}^{N_{1k}} \sum_{j=1}^{N_{2k}} O(\deg(x_{ki}) * \deg(y_{kj}) * \max\{\deg(T_1), \deg(T_2)\} * \log(\max\{\deg(T_1), \deg(T_2)\}))$$

$$\leq O(|T_1| * |T_2| * \max\{\deg(T_1), \deg(T_2)\} * \log(\max\{\deg(T_1), \deg(T_2)\})) \quad (4)$$

(da  $\sum_{k=1}^M \sum_{i=1}^{N_{1k}} \deg(x_{ki}) < |T_1|$  und  $\sum_{k=1}^M \sum_{j=1}^{N_{2k}} \deg(y_{kj}) < |T_2|$ )

Kombiniert man nun die Kosten für (1) und (2) so erhält man die Kosten (4) für die Gesamtlaufzeit dieser Phase.

#### 4. Generierung eines kostenminimalen Bearbeitungsskriptes:

In dieser Phase werden alle Knoten in  $T_1$  und  $T_2$  genau einmal traversiert, also erhält man die Kosten  $O(|T_1| + |T_2|)$  für diese Phase.

Anhand der Laufzeit der einzelnen Phasen läßt sich erkennen, daß die für die Laufzeit des Algorithmus entscheidende Phase Phase 3 ist und somit ergibt sich die Laufzeit (4) für den gesamten Algorithmus.

## Implementierung

Der X-Diff Algorithmus ist in C++ implementiert. Eine Java-Version des Algorithmus ist in Arbeit.

Der Algorithmus bekommt zwei XML Dokumente als Eingabe. Zuerst wird der IBM XML for C++ Parser (XML4C) benutzt um die beiden Dokumente in DOMBäume zu überführen. Als nächstes werden für jeden Knoten sein XHashwert und seine Signatur berechnet. Nun werden die Wurzelknoten beider Bäume auf Gleichheit getestet. Sind beide Bäume unterschiedlich, so berechnet X-Diff ein kostenminimales matching. Basierend auf diesem matching wird ein *delta* Baum (repräsentiert die Veränderungen zwischen beiden Dokumenten) erstellt, in dem die Knoten mit „Insert“, „Delete“ und „Update“ gekennzeichnet sind.

## Quellenverzeichnis

- X-Diff – Detecting changes in XML Documents  
Paper und Algorithmus:  
<http://www.cs.wisc.edu/~yuanwang/xdiff.html>
- Paper:  
Detecting Changes in XML Documents von G. Cobéna, S. Abiteboul und A.Marian
- Xerces Parser:  
<http://xml.apache.org>
- XMLTreeDiff:  
<http://www.alphaworks.ibm.com/tech/xmltreediff>
- XML for C++ Parser:  
<http://www.alphaworks.ibm.com/tech/xml4c>