XML Document Correction: Incremental Approach Activated by Schema Validation

Béatrice Bouchou Ahmed Cheriat Mírian Halfeld Ferrari Agata Savary

{beatrice.bouchou, ahmed.cheriat, mirian.halfeld, agata.savary}@univ-tours.fr

Université François Rabelais Tours – LI/Campus de Blois– 3 place Jean Jaurès – 41000 Blois, France

Abstract

Updating XML documents submitted to schema constraints requires incremental validation, i.e. checking the parts of the document concerned by the updates. We propose to correct subtrees for which re-validation fails: if the validator fails at node p, a correction routine is called in order to compute corrections of the subtree rooted at p, within a given threshold. Then validation continues. In the correction process, we limit ourselves to single typed tree language. The correction routine uses tree edit distance matrices. Different correction versions are proposed to the user.

1. Introduction

The validation of an XML document *w.r.t* schema constraints consists in testing whether the document verifies a set of structural specifications. Supposing that updates are applied to the document, an incremental validator is the one that verifies whether the updated document complies with the schema, by validating only the parts of the document involved in the updates.

We associate the validation process with correction proposals. During the execution of our validation method, if a constraint violation is found, a correction routine is called in order to propose local solutions capable of allowing the validation process to continue. Below we give an overview of our method.

The incremental validation approach

The validator takes a tree automaton representing a single typed tree grammar ([12]), an XML document and a sequence of updates to be performed on the document. This sequence is treated as one unique transaction. The algorithm checks whether the updates should be accepted or not. It proceeds by treating the XML tree in the document order.

Example 1 The XML tree of Fig. 1 is valid w.r.t. a given schema constraints. Update positions are marked. Suppose that the tree τ_1 being inserted is *locally valid*. When an open tag concerning an update position is reached, the incremental validation procedure takes the update operation into account (by deleting or inserting a subtree). Validation tests are performed only on parent nodes p of update positions $p.i$ (*i.e.* when the close tag of p is reached). On Fig. 1, a validation test is performed at position 0, due to the *delete* at 0.1, and at position ϵ due to the *insert* at 1. □

In this paper, a tree automaton is used to represent schema constraints (expressed in XML Schema). The incremental validation algorithm is a simplified version of the one proposed in [4]. Each validation step corresponds to checking whether a word w is in a given language $L(E)$, where w results from the concatenation of the states associated to p's children and E is a regular expression defining the structure that p's children should respect.

The correction approach

If the validation test fails at node p, a correction routine is called. The goal of this routine is to correct the invalid subtree rooted at p. The correction routine assumes that p's label l is a correct one and considers corrections over its descendants. To build a new and valid subtree with root l, the correction algorithm can propose changes on p's children. These modifications may assume changes on p's grandchildren and so forth, until reaching the leaves of the subtree.

To correct an invalid subtree at position p whose root label is assumed to be l, (*i.e.* to obtain a valid subtree whose root is labeled by l) our algorithm proposes changes that consist in: (i) changing labels of some p's descendants; (ii) deleting subtrees of the subtree rooted at p or (iii) inserting subtrees in the subtree rooted at p. In other words, given an invalid subtree T_p, we consider *update operations* capable of transforming T_p into a valid subtree T'_p. However, we want T'_p to be *close* to T_p. This means that a cost is assigned to each update operation and that the total cost of transforming T_p into T'_p is limited by a given threshold.

Example 2 The regular expressions assigned to labels in the tree grammar (a DTD) corresponding to Fig. 1 are those in Fig. 2.

Notice that f and g are elements associated to $data$. As seen in Ex. 1, during the incremental validation procedure the deletion at

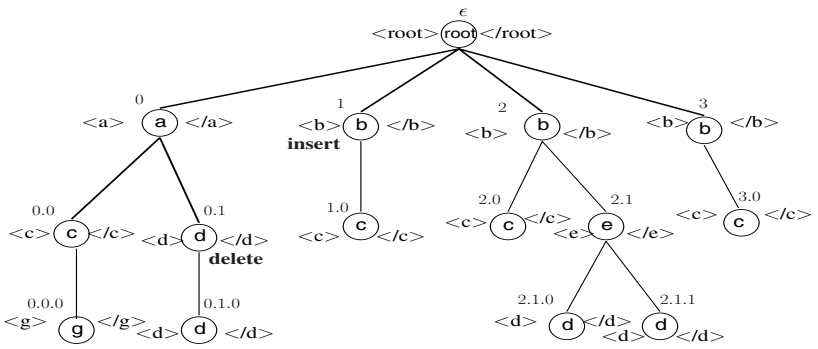

Figure 1. XML tree and update operations: Deletion of subtree rooted at position 0.1 and insertion of tree τ_1 at position 1. A node label a corresponds to an open tag `<a>` and a close tag `</a>` in the XML document.

Label	Regular expression	Label	Regular expression	Label	Regular expression
$root$	ab^*	c	$g^*f?$	e	d^*
a	$(cd)^* \mid m^*$	d	d^*	f	$data$
b	ce^*	g	$data$	m	g

Figure 2. Regular Expressions of a DTD

position 0.1 is considered and then the validation test performed at node 0 fails. Let $w = c$ be the word composed by the concatenation of the children of position 0, after the deletion. Clearly $w \notin L(E_a)$. The first possible correction is obtained by reinserting[1] d at position 0.1 with cost 1[2]. The second correction can be built by relabeling position 0.0 by m with cost 1 (as m also requires a child labeled g). The third correction consists in deleting the subtree rooted at position 0.0. The total cost of this operation is 2 since it corresponds to the removal of two nodes (those at positions 0.0 and $0.0.0$). Within an error threshold $th = 2$, all three corrections can be proposed as possible solutions. □

Our algorithm produces different local solutions for each invalid subtree. At the end of the re-validation process, global solutions are proposed to the user who can choose the best one for his update purposes.

Related work

The tree-to-tree correction problem has been formalized as an extension of the string-to-string correction problem [16]. In [5], it is pointed out that the diversity of the possible choices of elementary editing operations, which is important for the algorithm's complexity, is very large in case of a tree, as one can consider changes not only on the siblings' level but also on some ancestors level. The most appropriate choice depends clearly on the intuitive notion of tree proximity for each particular application.

Selkow [14] extends the *string edit distance* computation to *tree edit distance* computation, considering the following elementary edit operations: *relabel* a node, *delete* or *insert* a leaf node. By combining them, he can deal with insertion and deletion of entire subtrees. A cost equal to 1 is assigned to each elementary operation. To compute the distance between trees T and T', an edit distance matrix is built, in which each element $[i, j]$ contains the distance between the partial tree $T\langle i\rangle$ of T and the partial tree $T'\langle j\rangle$ of T', where $T\langle i\rangle$ is equal to T without its last subtrees, rooted from $i+1$ to its rightmost child. As our proposition is built upon the same idea, the reader can consider definitions and examples in Section 3 for details.

In [15] the authors consider different primitive operations and define *mappings*, that are similar to traces in the string edit distance problem. The minimum edit distance between T and T' is computed by looking for minimum cost mappings from T to T'. This approach is simplified and extended in [17]: the same basic definitions and assumptions held, but only some nodes are compared (called *keyroots*). Notice that all these proposals only consider *edit distance* computation.

In [6], the problem of *tree correction* is addressed. More precisely, given a tree T and a tree grammar G such that $T \notin L(G)$, $L(G)$ being the tree language defined by G, they propose an algorithm to find another tree $T' \in L(G)$ having a minimal edit distance from T. As in our proper

[1] Notice that although this solution is considered in the current version of our method, one may decide to assign a low priority to it since it corresponds to the rejection of the update.

[2] Another potential correction consisting of reinserting d both at 0.1 and at $0.1.0$ is not admitted as, in our method, insertions are limited to minimal-cost subtrees having the required label (here d) as root.

framework, trees represent XML documents and the grammar represents a DTD. The algorithm works on a binary representation of trees. It considers the following elementary edit operations: *relabel* of a node, *insertion* and *deletion* of a node *in the binary tree*, with assumptions on the deletion (it can occur only if at least one child is a leaf). The method is linear in the number of nodes and exponential in the number of errors: a given value prevents to compute T' if the number of errors is too large.

Our correction algorithm runs directly on unranked trees, in a context which differs from the one considered in [6], *i.e.* it is called during an incremental validation step, when updates are considered on a valid document, and it computes several correction candidates, together with tree edit operation sequences leading to these candidates. As we shall show, it generalizes both proposals of [14] and [13].

2 Preliminaries

A tree t over an alphabet Σ is a function $t : dom(t) \rightarrow \Sigma \cup \{\lambda\}$. Each element from $dom(t)$ is a sequence of integers representing a node position of tree t and each node of position p is labeled with the symbol $t(p)$. The root is represented by the empty sequence ϵ and the empty tree is defined as $t = \{(\epsilon, \lambda)\}$. Let $\mathbb{N}^*$ be the set of finite words over the set of natural numbers $\mathbb{N}$ and λ be a special label. The domain $dom(t) = \{p \in \mathbb{N}^* \mid (p, l) \in t\}$ is closed under prefixes[3] and satisfies the following property: for $j \in \mathbb{N}$ and $u \in \mathbb{N}^*$; if $u.j \in dom(t)$, then for all $0 \leq i \leq j$, $u.i \in dom(t)$. The set of *leaves* of t is defined by $leaves(t) = \{u \in dom(t) \mid \neg\exists i \in \mathbb{N}$ such that $u.i \in dom(t)\}$. Fig. 1 represents a tree whose alphabet is the set of element names appearing in an XML document.

Given a tree t we denote by t_p the subtree whose root is at position $p \in dom(t)$, *i.e.* $t_p = \{(r, t(r)) \mid r = p.u$ and $r \in dom(t), u \in \mathbb{N}^*\}$. Note that a subtree is not a tree (its root is not at ϵ), thus we define a tree t_p^{tree} resulting from subtree t_p as follows: $dom(t_p^{tree}) = \{s \mid p.s \in dom(t)$ and $s \in \mathbb{N}^*\}$ and for each $s \in dom(t_p^{tree})$ we have $t_p^{tree}(s) = t(p.s)$. For instance, in Fig. 1 $t_0 = \{(0, a), (0.0, c), (0.0.0, g), (0.1, d), (0.1.0, d)\}$. Thus, we have $t_0^{tree} = \{(\epsilon, a), (0, c), (0.0, g), (1, d), (1.0, d)\}$.

XML documents are seen as ordered unranked labeled trees that should respect some schema constraints that are expressed by the transition rules of a tree automaton. We define a *tree automaton* over an alphabet Σ as a tuple $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ where Q is a set of states, $Q_f \subseteq Q$ is a set of final states and Δ is a set of transition rules of the form $a, E \rightarrow q$ where *(i)* $a \in \Sigma$; *(ii)* E is a regular expression over Q and *(iii)* $q \in Q$. In this paper, to simplify notation, we restrict the presentation to DTDs, which are local tree grammars [12]. It means that the corresponding automaton associates only one state to each label, so we consider[4] that $Q = \Sigma$. Fig. 2 shows the transition rules of a tree automaton $\mathcal{A}$ imposing constraints over the document of Fig. 1. In this case, $Q_f = \{root\}$.

A tree may be changed through one or more update operations. In this paper, an update operation is seen as a high level operation defined over simple edit operations. Given a tree t, an *edit operation* may be applied to an *edit position* p provided that p respects some constraints depending on the type of the edit operation.

Definition 1 - *Edit Operations:* Let ed be a tuple (op, p, l) where: $op \in \{add, remove, relabel\}$; p an edit position and l a label in Σ or null ($/$). Given a tree t, an edit operation is a partial function that transforms the tree t into a tree t' ($t \xrightarrow{ed} t'$):

- *relabel* changes the label associated with p;
- *remove* allows the removal of a leaf and
- *add* allows the addition of a single leaf at any of the existing positions or "around" them, except at the root of a non-empty tree.

For edit operations on position ϵ, for all $l_1, l_2 \in \Sigma \cup \{\lambda\}$, we define $\{(\epsilon, l_1)\} \xrightarrow{(relabel,\epsilon,l_2)} \{(\epsilon, l_2)\}$ and $\{(\epsilon, l_1)\} \xrightarrow{(remove,\epsilon,/)} \{(\epsilon, \lambda)\}$, and $\{(\epsilon, \lambda)\} \xrightarrow{(add,\epsilon,l)} \{(\epsilon, l)\}$. All other edit operations are undefined. □

Example 3 Consider the XML tree of Fig. 1: the domain $dom(T')$ of the resulting tree T' obtained after applying edit operation $ed = (add, 1, a)$ is $dom(T') = \{\epsilon, 0, 0.0, 0.1, 0.0.0, 0.1.0, 1, 2, 2.0, 3, 3.0, 3.1, 3.1.0, 3.1.1, 4, 4.0\}$. The new tree T' is shown on Fig. 3(a). □

Definition 2 - *Update Operations:* An update operation upd is a tuple (op, p, τ) where: $op \in \{insert, delete, replace\}$, p is a position, and τ is a tree that must be empty for a $delete$. Each update operation corresponds to a sequence (composition) of edit operations.

- The insertion of a tree τ at position p in a tree t is performed by adding each node of τ (one by one). To minimize the number of shifts, we add nodes of τ from its root to its leaves, and from left to right.
- The deletion of the subtree rooted at position p in a tree t is performed by removing all its nodes one by one, from leaves to root and from right to left.
- The replacement of the subtree at position p in a tree t by a tree τ is defined as the minimum sequence of add, $remove$, and $relabel$ operations necessary to transform t_p^{tree} into τ. □

[3] The prefix relation in $\mathbb{N}^*$, denoted by $\preceq$ is defined by: $u \preceq v$ iff $u.w = v$ for some $w \in \mathbb{N}^*$. A set $dom(t) \subseteq \mathbb{N}^*$ is closed under prefix if $u \preceq v, v \in dom(t)$ implies $u \in dom(t)$.

[4] In case of single typed tree languages we implement the automaton run in such a way that one single state is associated to each node, so the presentation is obviously generalized to grammars defined with XML Schema.

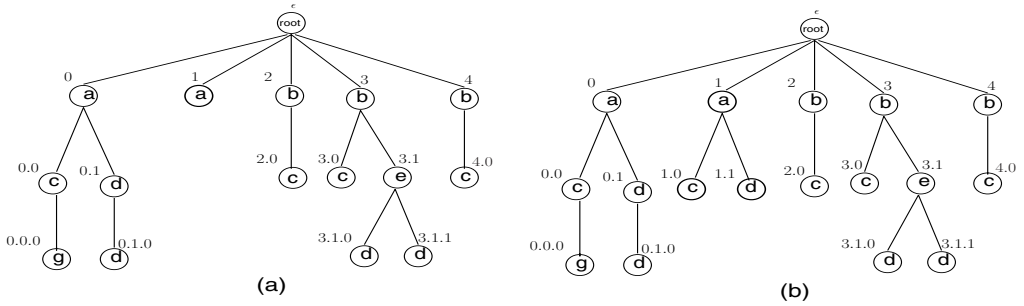

Figure 3. Application of (a) the edit operation $(add, 1, a)$, (b) the update operation $(insert, 1, \tau_1)$ over Fig. 1.

Example 4 Let T be the tree of Fig. 1 and $(insert, 1, \tau_1)$ the update operation with $\tau_1 = \{(\epsilon, a), (0, c), (1, d)\}$. This update operation is translated into the sequence $(add, 1.\epsilon, a)$, $(add, 1.0, c)$, $(add, 1.1, d)$. The resulting tree is shown on Fig. 3(b). □

For any edit operation ed, we define $cost(ed) = 1$ to be the cost of performing ed. Given an update operation $t \xrightarrow{upd} t'$, equivalent to the sequence $t = t_0 \xrightarrow{ed_1} t_1 \xrightarrow{ed_2} t_2 \cdots \xrightarrow{ed_n} t_n = t'$ the cost of upd is $Cost(upd) = \Sigma_{i=1}^{n}(cost(ed_i))$. We will show in Section 3 that the cost for a replacement of t_p by t'_p corresponds to the minimal distance between t_p and t'_p. We generalize the concept of update operation cost to introduce the cost of a sequence of update operations. Thus we note $t \xrightarrow{(updSeq)} t'$, to indicate $t = t_0 \xrightarrow{upd_1} t_1 \xrightarrow{upd_2} t_2 \cdots \xrightarrow{upd_m} t_m = t'$ and we define $Cost(updSeq) = \Sigma_{i=1}^{m}(cost(upd_i))$.

We can now define the notion of distances between two trees and between a tree and a tree language.

Definition 3 - *Tree distances:* Let t and t' be trees. Let $\mathcal{S}$ be the set of all update sequences capable of transforming t into t'. The distance between t and t' is defined by: $dist(t, t') = \min_{S_i \in \mathcal{S}} \{Cost(S_i)\}$. The distance between a tree t and a tree language $\mathcal{L}$ is defined by: $DIST(t, \mathcal{L}) = \min_{t' \in \mathcal{L}} \{dist(t, t')\}$. □

3 Correcting an invalid tree

Let $\mathcal{L}$ be the tree language defined by the tree automaton $\mathcal{A}$. Let $\mathcal{L}_l \subseteq \mathcal{L}$ be the tree language defined by transition rules in $\mathcal{A}$ that contains all trees whose root is labeled with l.

Given an XML tree T, we assume that the validation fails at position p whose label is l (*i.e.* $T(p) = l$ and $T_p^{tree} \notin \mathcal{L}_l$) and we propose a routine capable of correcting T_p. Our correction routine assumes that p has the correct label l and considers changes on p's descendants. It takes the language $\mathcal{L}_l$, the tree $T_p^{tree} \notin \mathcal{L}_l$ and an error threshold as input. It looks for new trees $T_p^{tree'} \in \mathcal{L}_l$ such that $dist(T_p^{tree}, T_p^{tree'})$ is within the threshold. Each new tree $T_p^{tree'}$ can then be translated into a new subtree T'_p for T.

Example 5 In Example 2 our validation procedure fails at position 0 of the XML tree of Fig. 1. At this point our correction routine can be activated. To perform corrections this routine considers the tree language $\mathcal{L}_a$ defined by the tree automaton whose transition rules are those shown in Fig. 2 but with $Q_f = \{a\}$. Clearly, as $T_0^{tree} = \{(\epsilon, a), (0, c), (0.0, g)\}$, we have $T_0^{tree} \notin \mathcal{L}_a$. □

In the rest of this paper, in order to simplify notation, we use t to denote the tree T_p^{tree} and t' to denote the tree $T_p^{tree'}$.

Our algorithm extends the ones proposed both in [14] and in [13]: it uses an edit distance matrix H_DIST. Each element $H\_DIST[i, j]$ contains the edit distance between *partial* trees $t\langle i \rangle$ and $t'\langle j \rangle$, as well as the set of update sequences capable of transforming $t\langle i \rangle$ into $t'\langle j \rangle$ with a minimal cost. A (partial) tree $t\langle i \rangle$ is composed by the root of t and its subtrees $t_0, \ldots t_i$ (Fig. 4). Obviously, if the fanouts of the roots of t and t' are n and m, respectively, then $t\langle n \rangle = t$ and $t'\langle m \rangle = t'$. The matrix is calculated column per column. Each new element is deduced from its three top-left-hand neighbor elements which have been calculated previously ($H\_DIST[i-1, j]$, $H\_DIST[i, j-1]$ and $H\_DIST[i-1, j-1]$). Each $H\_DIST[i, j]$ stores a cost and a set of update sequences obtained from its neighbors and from the updates that are necessary to: delete the subtree t_i (Fig. 4(b) edge (3) in the matrix), insert a new subtree t'_j (Fig. 4(b) edge (1)), or replace the subtree t_i by a new subtree t'_j (Fig. 4(b) edge (2)).

The computation of each $H\_DIST[i, j]$ follows a "horizontal-vertical" reasoning. Recall that t is the tree to be corrected and t' is a correction candidate we have to construct so that its root's children respect the regular expression $E_{t(\epsilon)}$. We consider the word $w \notin L(E_{t(\epsilon)})$ composed by the concatenation of the states associated to the children of t's root. On the "horizontal" plan, our correction algorithm follows the one of [13], *i.e.* it tries to build new words $w' \in L(E_{t(\epsilon)})$, as close as possible to w via a threshold-controlled depth-first exploration of the Finite

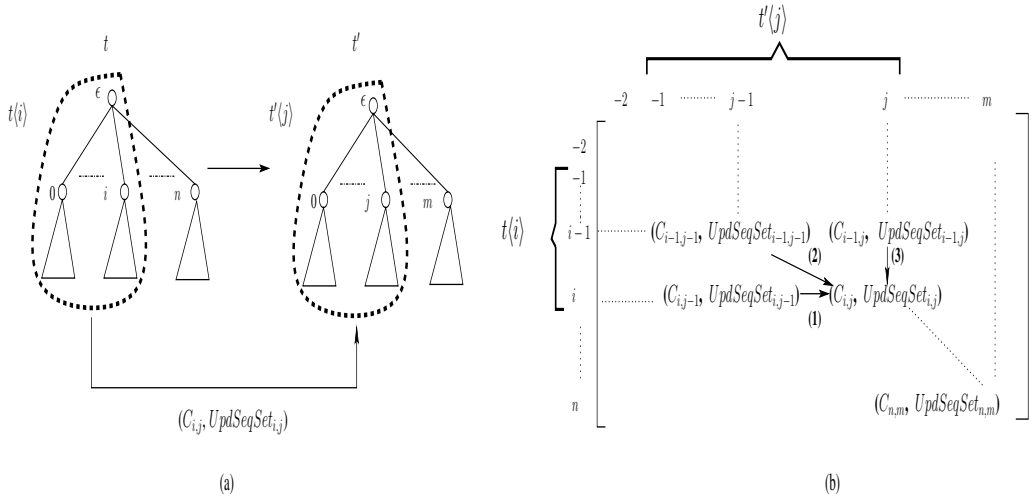

Figure 4. **(a) Two (partial) trees $t\langle i\rangle$ and $t'\langle j\rangle$. (b) Matrix H_DIST for trees t and t', and computation of $H\_DIST[i,j]$.**

State Automaton (FSA) corresponding to $L(E_{t(\epsilon)})$. Each time a transition is followed a new column is added to the distance matrix. If a final state is reached and the current bottom-right-hand element of the matrix does not exceed a given threshold then the current word w' is a valid correction candidate. Otherwise we may either continue to the next transition (if any element of the current column does not exceed the threshold) or backtrack, delete the current column, and explore a different path.

Since in our approach each "character" in w is the root of a subtree, in order to obtain w', we also have to work in the "vertical" direction. Indeed, each character w'_j has to yield a tree $t'_j \in \mathcal{L}_{t'(j)}$, with root label $t'_j(\epsilon) = w'[j] = t'(j)$.

Therefore, while on the "horizontal" plan we deal with a matrix that associates a word w to a regular language, on the "vertical" plan, we deal with a subtree t_i, in a way similar to the computation of tree edit distance in [14]. In order to correct t_i, not only do we have to build a matrix for its root's sons, but possibly also for the grandchildren, and so on. To conclude, as formalized in the following definition, matrix H_DIST stores the edit distance between the invalid tree t that we are trying to correct and its partial correction candidates, together with the set of update sequences necessary to transform t into each of the candidates.

Definition 4 ***- Edit Distance Matrix:*** Let t be a tree and let $\mathcal{L}$ ($\mathcal{L} = \mathcal{L}_{t(\epsilon)}$) be a tree language such that $t \notin \mathcal{L}$. An edit distance matrix, denoted $H\_DIST_{t,t'}$ (or just H_DIST when no confusion is possible), stores the edit distance between tree t and a partial tree $t' \in \mathcal{L}$. Let n and m be the fan-outs of the roots of t and t', respectively.
The matrix H_DIST is a two dimensional matrix with indices starting from -2. Each element $H\_DIST[i,j] = (C, updSeqSet)$ where C is a cost and ***updSeqSet*** is a set of update sequences transforming $t\langle i\rangle$ into $t'\langle j\rangle$ and having cost C, as defined below.

1. Initializations

- For each row i: $H\_DIST[i,-2] = (\infty, \top)$ and for each column j: $H\_DIST[-2,j] = (\infty, \top)$, where $\top$ is a theoretical infinite update sequence of cost ∞.

- Element $H\_DIST[-1,-1]$ is calculated as follows :

$$\begin{array}{ll} (0, \{[]\}) & \textbf{if } t(\epsilon) = t'(\epsilon) \\ (1, \{[(relabel, \epsilon, t'(\epsilon))]\}) & \textbf{if } t(\epsilon) \neq \lambda,\ t'(\epsilon) \neq \lambda \\ & \text{and } t(\epsilon) \neq t'(\epsilon) \\ (1, \{[(add, \epsilon, t'(\epsilon))]\}) & \textbf{if } t(\epsilon) = \lambda \text{ and} \\ & t(\epsilon) \neq t'(\epsilon) \\ (1, \{[(remove, \epsilon, /)]\}) & \textbf{if } t'(\epsilon) = \lambda \text{ and} \\ & t(\epsilon) \neq t'(\epsilon) \end{array}$$

2. For all $j \geq -1$, and for all $i \geq -1$, with $(i,j) \neq (-1,-1)$:

- $C = min(C_{replace}, C_{insert}, C_{delete})$ with

  $- C_{replace} := H\_DIST[i-1, j-1].C + dist(t_i, t'_j)$

  $- C_{insert} := H\_DIST[i, j-1].C + dist(\{(\epsilon, \lambda)\}, t'_j)$

  $- C_{delete} := H\_DIST[i-1, j].C + dist(t_i, \{(\epsilon, \lambda)\})$

- $updSeqSet = \{Seq \mid Seq \in \mathcal{S}_{replace} \cup \mathcal{S}_{insert} \cup \mathcal{S}_{delete}$, $Cost(Seq) = C\}$ where:

  $-$ $\mathcal{S}_{replace}$ is the Cartesian product of *(i)* the set of update sequences transforming $t\langle i-1\rangle$ into $t'\langle j-1\rangle$ and *(ii)* the set of update sequences transforming t_i into t'_j. Formally, $\mathcal{S}_{replace} = \mathcal{S}'_r \times \mathcal{S}''_r$ where:

  $\star$ $\mathcal{S}'_r = H\_DIST[i-1, j-1].updSeqSet$

  $\star$ $\mathcal{S}''_r = \{Seq'' \mid t_i \overset{(Seq'')}{\longrightarrow} t'_j$ and $Cost(Seq'') = dist(t_i, t'_j)\}$

$-$ $\mathcal{S}_{insert}$ is the Cartesian product of *(i)* the set of update sequences transforming $t\langle i\rangle$ into $t'\langle j-1\rangle$ and *(ii)* the set of update sequences inserting t'_j. Formally, $\mathcal{S}_{insert} = \mathcal{S}'_i \times \mathcal{S}''_i$ where:

$\star$ $\mathcal{S}'_i = H\_DIST[i, j-1].updSeqSet$

$\star$ $\mathcal{S}''_i = \{Seq'' \mid \{(\epsilon,\lambda)\} \stackrel{(Seq'')}{\longrightarrow} t'_j$ and $Cost(Seq'') = dist(\{(\epsilon,\lambda)\}, t'_j)\}$

$-$ $\mathcal{S}_{delete}$ is the Cartesian product of *(i)* the set of update sequences transforming $t\langle i-1\rangle$ into $t'\langle j\rangle$ and *(ii)* the set of update sequences deleting t_i. Formally, $\mathcal{S}_{delete} = \mathcal{S}''_d \times \mathcal{S}'_d$ where:

$\star$ $\mathcal{S}'_d = H\_DIST[i-1, j].updSeqSet$

$\star$ $\mathcal{S}''_d = \{Seq'' \mid t_i \stackrel{(Seq'')}{\longrightarrow} \{(\epsilon,\lambda)\}$ and $Cost(Seq'') = dist(t_i, \{(\epsilon,\lambda)\})\}$ □

Notice that the first column of H_DIST ($H\_DIST[i,-2]$) stores very great costs (∞) and the second one ($H\_DIST[i,-1]$) stores the cost and update sequences needed to delete the subtrees $t_0, \ldots t_i$. In a similar way, the first row contains very great costs and the second one ($H\_DIST[-1,j]$) contains the cost and update sequences needed to insert the subtrees $t'_0, \ldots t'_j$. According to [14], the bottom right-hand position of a matrix H_DIST contains the edit distance between t and $t' \in \mathcal{L}$. In our approach it is accompanied with all minimal cost update sequences transforming t into t'.

Example 6 In Example 5 the validation procedure fails at position 0 Correction routine is called for T_0^{tree} with $\mathcal{L}_a$ and $th = 2$. Let $FSA_E = (\{1,2\}, \{m,d,c\}, 1, \{\delta(1,m) = 1, \delta(1,c) = 2, \delta(2,d) = 1\}, \{1\})$ be the finite-state automaton corresponding to the regular expression of label a.

Step 1: The tree t being considered for correction is $\{(\epsilon,a),(0,c),(0.0,g)\}$. The matrix H_DIST_1 is initialized as specified in Definition 4. The initial (and finite) state 1 in FSA_E becomes the current state. Column -2 imposes a matrix border, necessary in the computation. Line -1 represents the root node a whose children are represented by lines $i \geq 0$ of H_DIST_1 .

As we assume that the root label of T_0^{tree} does not change (it stays as a), element $H\_DIST_1[-1,-1] = (0, \{[]\})$. In this case, the lowest cost operation is to compute $H\_DIST_1[0,-1]$ from $H\_DIST_1[-1,-1]$: assume the deletion of the subtree rooted at position 0 of T_0^{tree}. This deletion is treated by calling recursively the correction routine (*i.e.* by starting Step 2). Inputs are: the tree obtained from the subtree rooted at position 0 of T_0^{tree} and the tree language $\mathcal{L}_\lambda$ containing only the empty tree.

Step 2: We start the construction of a new matrix. The tree t being considered for correction is $\{(\epsilon,c),(0,g)\}$. The matrix H_DIST_2 is initialized. Now $H\_DIST_2[-1,-1] = (1, \{(remove, \epsilon, /)\})$ because it is calculated by considering the removal of $t(\epsilon) = c$. Similarly to Step 1, element $H\_DIST_2[0,-1]$ is computed from $H\_DIST_2[-1,-1]$ by assuming the deletion of the subtree $\{(0,g)\}$ of t. This deletion is treated by calling recursively the correction routine (Step 3).

Step 3: The tree t being considered for correction is $\{(\epsilon,g)\}$. The matrix H_DIST_3 is initialized and as t has only one root node ($H\_DIST_3[-1,-1] = (1, \{(remove, \epsilon, /)\})$).

Return to Step 2: Coming back (from the recursive call) to H_DIST_2 the result obtained in Step 3 corresponds to the deletion of the node at position 0 of the tree t in Step 2 ($H\_DIST_2[0,-1] = (2, \{(remove, 0, /)(remove, \epsilon, /)\})$). The deletion of t's root is also considered($H\_DIST_2[-1,-1] = (1, \{(remove, \epsilon, /)\})$).

Return to Step 1: Coming back to H_DIST_1, the result obtained in Step 2 corresponds to the deletion of the subtree rooted at position 0.0. We have $H\_DIST_1[-1,-1] = (0, \{[]\})$ and $H\_DIST_1[0,-1] = (2, \{(remove, 0.0.0, /)(remove, 0.0, /)\})$. The current state in FSA_E is a final one and the bottom right-hand corner element of H_DIST_1 does not exceed the threshold. Thus, this element contains a valid correction candidate. We refer to [7] for more details on this example. □

4 Algorithms

The algorithm whose goal is to compute valid candidate trees within a given threshold is called *CorrectionRoutine*. It receives as input a tree T_p^{tree} issued from an XML tree T whose validation failed at p. It receives also the set $solStock$ of solutions calculated for previously corrected trees (if any), and returns the same set enlarged with solutions calculated for T_p^{tree}. *CorrectionRoutine* contains a recursive procedure, called *CorrectSubtree*, which receives in its first call the initial state s_0 of FSA_E, an empty word w', an empty list $LCand$, the set $solStock$ and the matrix H_DIST with its two first columns initialized. It returns its solutions in $LCand$, which is added to $solStock$ at the end of *CorrectionRoutine*. Both p and $solStock$ allow to avoid re-correcting the same subtrees (in function *AddNewCol*).

Procedure CorrectSubtree(t, p, w, w′, th, H_DIST, FSA_E, s, LCand, solStock)
Input
(i) t: $t = T_p^{tree} \notin \mathcal{L}_{t(\epsilon)}$
(ii) p: position in $dom(T)$ such that $T_p^{tree} = t$
(iii) w: invalid word (concatenation of states associated to t's root's children)
(iv) th: integer corresponding to the error threshold
(v) FSA_E: deterministic FSA for E appearing in $t(\epsilon)$, $E \rightarrow q_{t(\epsilon)}$
(vi) s: current state in FSA_E
(vii) $solStock$: set of tuples $(pos, LCand)$ for subtrees of T corrected before t.
Input/Output
(i) w': partial valid word (concatenation of states associated to root's children in t')
(ii) H_DIST: edit distance matrix between t and t'
(iii) $LCand$: set of tuples $(C, updSeqSet)$ with C the cost and *updSeqSet* the set of update sequences having cost C

1. **if** s is a final state in FSA_E and $H\_DIST[|w|-1, |w'|-1].C \leq th$ { // New candidate found within the threshold.
2. $LCand := SortIns(LCand, H\_DIST[|w|-1, |w'|-1])$ }
3. **for each** transition $\delta(s,q') = s'$ in FSA_E **do** {
4. $w' = w'.q'$

5. $H\_DIST := AddNewCol(t,\ p,\ w,\ w',\ th,\ H\_DIST, q',\ solStock)$
6. **if** $(Cuted(\ w, w', th, H\_DIST) \leq th)\ \{$
7. *CorrectSubtree*$(t, p, w, w', th, H\_DIST, FSA_E, s', LCand,\ solStock)$ $\}$
8. $H\_DIST := DelLastCol(H\_DIST)$
9. $w' := DelLastSymbol(w')$ $\}$ □

FSA_E is explored in the depth-first order. Each time a transition is followed, on the "horizontal plan", the current word w' is extended (line 4) and a new column is added to H_DIST (line 5). This means that, on the "vertical plan", t' is also extended. That allows to check if t' may still lead to a candidate remaining within the threshold. If it does, the path is followed via a recursive call (line 7), otherwise the path gets cut off. In each case the transition is finally backed off (lines 8 and 9) and a new transition outgoing from the same state is tried out. If we arrive at a final state and the distance from t' to t does not exceed the threshold, then t' is a valid candidate that is inserted to the list $LCand$ of all candidates found so far (lines 1 and 2).

Function $Cuted$ computes the cut-off edit distance [13] between t and t'. It checks if all elements of the current column (line 6)exceed the threshold. If they do, there is no chance for t' to be a partial correction within the threshold.

Before calling *CorrectSubtree*, *CorrectionRoutine* first uses procedure *intializeMatrix* to initialize the first two columns of H_DIST (Definition 4). It recursively calls *CorrectionRoutine* to compute the cost and the update sequence necessary to delete each subtree t_i, since deleting a subtree is equivalent to correcting it *w.r.t* the empty tree.

All the following columns are calculated by function *AddNewCol* (line 5) which deduces each matrix' element from its three upper left-hand neighbors as stated in Definition 4. In particular, in case of insertion, the subtree t'_j is not known in advance (although its root's label is). Thus its insertion cost and update sequence is the one needed to create a minimal tree valid *w.r.t* $\mathcal{L}_{t'(j)}$, which is equivalent to correcting an empty tree *w.r.t* $\mathcal{L}_{t'(j)}$ (by calling *CorrectionRoutine*). Similarly, in case of replacement, the cost and update sequences are the ones needed to correct t_i *w.r.t* $\mathcal{L}_{t'(j)}$ (by also calling *CorrectionRoutine*).

Example 7 Consider the XML tree T of Fig. 1 after the first update operation. For $t = T_0^{tree}$ *CorrectionRoutine* returns $LCand = \{(1, \{(add, 0.1, d)\})\ (1, \{(relabel, 0.0, m)\}),\ (2, \{(remove, 0.0.0, /)(remove, 0.0, /)\})\}$. The first element of $LCand$ (with cost 1) is found by computing matrix $H\_DIST_{\{(\epsilon,\lambda)\},\{(\epsilon,d)\}}$, the second one (with cost 1) by computing $H\_DIST_{T_{0.0}^{tree},\{(\epsilon,m)(0,g)\}}$, and the third one (with cost 2) is found by computing $H\_DIST_{T_{0.0}^{tree},\{(\epsilon,\lambda)\}}$. □

Theorem 1 *Let $\mathcal{L}$ be a single typed tree language and $t \notin \mathcal{L}$. The algorithm CorrectionRoutine is correct (all the candidates it computes respect a given threshold) and complete (it computes all the candidates $t' \in \mathcal{L}$ such that $dist(t, t')$ is minimal within a given threshold).*[5] □

It is worth noting that *CorrectionRoutine* calculates not only all the set of candidates $t' \in \mathcal{L}$ such that $dist(t, t')$ is minimal within a given threshold but also some non-minimal candidates that respect the threshold. The procedure is however not complete *w.r.t* the set containing all the candidates respecting the threshold. This is due to optimization aspects: when correcting a subtree T_p, our algorithm avoids "correcting" its subtrees that are already valid because they come from the initially valid tree and no updates have been performed on them. For instance, consider the first solution proposed in Example 1 where we keep label c for position 0.0. Our algorithm considers that there is nothing to correct under this node since subtree $T_{0.0}$ is valid. However, a complete approach would have to propose modifications in this subtree like, for example, the insertion of g at position 0.0.1 (considering that $E_c = g^* f?$).

At the end of the incremental validation, the relation $solStock$ stores correct solutions: each failure position p corresponds to a tuple $(p, LCand)$ in $solStock$. As some redundancies may exist between new candidates in $LCand$ and candidates in $solStock$, before proceeding to the integration of solutions, we eliminate from $solStock$ the set of tuples $\{(p, LCand) \in solStock \mid \exists p', p' \prec p \wedge (p', LCand') \in solStock\}$. This is because corrections at positions $p' \prec p$ take corrections at p into account. During the correction at p', if a candidate tree for $T_{p'}$ relies on corrections on T_p (which belongs to $T_{p'}$) then the list $LCand$ computed for p is integrated into $LCand'$ (for p').

Local solutions in $solStock$ are combined into global ones, each one having a global cost. If this cost does not exceed the threshold, then the corresponding solution is a candidate. The user can choose to compute one unique solution or n ones, depending on whether he is interested in any correction with a minimal distance from T or in examining several possible corrections.

Example 8 Consider Example 7. After the first correction at position 0 , $SolStock$ contains $S = (0, LCand_1)$, where $LCand_1$ represents the list of candidates found in Example 7. The second error is found when the validation test is performed at position ϵ: the resulting tree T is not in $\mathcal{L}_{root}$ where $\mathcal{L}_{root}$ is the tree language defined by the rules listed in Example 2 with $Q_f = \{root\}$. New corrections are then computed. When $th = 2$, *CorrectionRoutine* for $t = T_\epsilon^{Tree}$ (updated tree), returns $null$, since all candidate solution costs exceed the threshold. But if we consider $th = 3$, then five candidates are proposed. □

Global solutions can be presented to the user in different forms, *i.e.* either candidate *trees* or candidate *operation sequences*.

[5] The proof is given in [7].

5 Conclusions and perspectives

This paper introduces a tree-to-grammar correction approach: given a single typed tree language $\mathcal{L}$ (*e.g.* defined with XML Schema) and an invalid tree t, find valid trees t' whose distance from t is within a given threshold th. It extends our previous work in [9] which considers an incremental string-to-grammar correction method. Our theoretical complexity is exponential in the size of node fan-out in the document to be corrected (as in [9]).

Despite its high theoretical complexity, experimental results show the good performance of our *string-to-grammar correction algorithm* [8, 9]: by varying the regular expression, the threshold, the size of the initial word and the number of updates the time execution of the 90% fastest runs is less or equal than 44 ms. We also have good experimental results for the *incremental validation routine* [4]: it takes *almost a third* of the time needed for an efficient commercial product (running from scratch) to incrementally validate 50 updates on a document having $61,000,000$ nodes. The implementation of our complete *tree-to-grammar correction algorithm* is in progress (based on [9, 14]).

Novelties of our approach:

Integration of correction and validation process: We not only perform a validation but we also propose corrections at positions where the validation fails. Most existing XML tools include either only validators ([2], [3]), or correctors limited to the well-formedness of the XML-tree ([1]).

Incremental validation and correction: Parts of the tree not concerned by the updates are omitted during validation and correction, allowing better time and space performances. Experiments on various versions of correction (incremental and local / from scratch and global) remain to be performed in order to confirm this intuitively obvious hypothesis.

An extension of a whole branch of finite-state word-to-language correction algorithms: Our method extends correction algorithms such as those in [13], [11] into the tree-to-language correction problem.

Some extension directions for our algorithm:

Performance of large-scale tests: They are to be performed in order to experimentally determine its performances.

Extension of update operations: One may need insertion or deletion of an internal node, or a subtree move, as an elementary operation. Inserting or deleting an internal node should rely, in our approach, on a local word-to-language correction model in which one letter can be replaced, in one operation, by several letters, and conversely. That is because deleting an internal node is equivalent to shifting some nodes up, and thus to replacing the label of the deleted node by the sequence of labels of its sons. Conversely, inserting an internal node relies on shifting some nodes one level down, and thus on replacing the sequence of their labels by the label of the new node. Some effort has been done for such an extended word correction (e.g. [10]) in which sequence-to-sequence edit operations are admitted as elementary, however we think that an elegant model of this problem is yet to be achieved.

Introduction of a complete XML correction framework: This framework might concern attributes as well as integrity constraints. We already deal with all these aspects for incremental validation ([4]).

References

[1] NXML. At http://www.thaiopensource.com/nxml-mode/.

[2] XMLmind. At http://www.xmlmind.com/xmleditor/.

[3] XMLSpy. At http://www.altova.com/download/.

[4] M. A. Abrao, B. Bouchou, A. Cheriat, M. Halfeld Ferrari, D. Laurent, and M. A. Musicante. Incremental Constraint Validation of XML Documents Under Multiple Updates. In *Submitted as journal paper*, 2005.

[5] D. T. Barnard, G. Clarke, and N. Duncan. Tree-to-tree Correction for Document Trees. Technical Report 95-372, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1995.

[6] U. Boobna and M. de Rougemont. Correctors for XML Data. In *Database and XML Technologies, Second International XML Database Symposium*, volume 3186 of *LNCS*, pages 97–111. Springer-Verlag, 2004.

[7] B. Bouchou, A. Cheriat, M. Halfeld Ferrari, and A. Savary. Integrating Correction into Incremental Validation. Technical Report 292, Université de Tours, LI/ Campus de Blois. Also in BDA'06, 2006.

[8] B. Bouchou, A. Cheriat, A. Savary, and M. Halfeld Ferrari. Incremental String Correction with respect to a Finite-State Grammar. Technical Report 282, Université de Tours, LI/ Campus de Blois, 2005.

[9] A. Cheriat, A. Savary, B. Bouchou, and M. Halfeld Ferrari. Incremental String Correction: Towards Correction of XML Documents. In *Prague Stringology Conference*, 2005.

[10] S. Deorowicz and M. Ciura. Correcting spelling errors by modelling their causes. *International Journal of Applied Mathematics and Computer Science*, 15(2):275–285, 2005.

[11] S. Mihov and K. U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477, 2004.

[12] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML Schema Language using Formal Language Theory. In *ACM, Transactions on Internet Technology (TOIT)*, 2004.

[13] K. Oflazer. Error-tolerant Finite-state Recognition with Applications to Morphological Analysis and Spelling Correction. *Computational Linguistics*, 22(1):73–89, 1996.

[14] S. M. Selkow. The Tree-to-Tree Editing Problem. *Information Processing Letters*, 6(6):184–186, 1977.

[15] K.-C. Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery*, 26(3), 1979.

[16] R. A. Wagner and M. J. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168–173, 1974.

[17] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problem. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.