On Correcting XML Documents With Respect to a Schema

Journal:	*The Computer Journal*
Manuscript ID:	COMPJ-2012-01-0058.R3
Manuscript Type:	Original Article
Date Submitted by the Author:	n/a
Complete List of Authors:	Amavi, Joshua; Université d'Orléans, LIFO Bouchou, Béatrice; Université François Rabelais Tours, Laboratoire d'Informatique (LI) Savary, Agata; Université François Rabelais Tours, Laboratoire d'Informatique (LI)
Key Words:	XML Processing, Document-to-Schema Correction, Tree Edit Distance

On Correcting XML Documents With Respect to a Schema

JOSHUA AMAVI[1], BÉATRICE BOUCHOU[2] AND AGATA SAVARY[2]

[1]*LIFO - Université d'Orléans, Orléans, France*
[2] *Université François Rabelais Tours, LI, Blois Campus, France*
Email: beatrice.bouchou@univ-tours.fr

We present an algorithm for the correction of an XML document with respect to schema constraints expressed as a DTD. Given a well-formed XML document t seen as a tree, a schema S and a non negative threshold th, the algorithm finds every tree t' valid with respect to S such that the edit distance between t and t' is no higher than th. The algorithm is based on a recursive exploration of the finite-state automata representing structural constraints imposed by the schema, as well as on the construction of an edit distance matrix storing edit sequences leading to correction candidate trees. We prove the termination, correctness and completeness of the algorithm, as well as its exponential time complexity. We also perform experimental tests on real-life XML data showing the influence of various input parameters on the execution time and on the number of solutions found. The algorithm's implementation demonstrates polynomial rather than exponential behavior. It has been made public under the GNU LGPL v3 license. As we show in our in-depth discussion of the related work, this is the first full-fledged study of the document-to-schema correction problem.

Keywords: XML Processing; Document-to-Schema Correction; Tree Edit Distance

Received 00 January 2012; revised 00 Month 2012

1. INTRODUCTION

The correction of an XML document t *w.r.t.* a set of schema constraints S consists in computing new documents that verify the set of structural specifications stated in S and that are close to t. Applications of this problem are important and vary widely, as extensively shown in [1], and include:

- XML data exchange and integration,
- web service searching and composition,
- adapting an XML document *w.r.t.* a database [2, 3],
- performing consistent queries on inconsistent XML documents [4],
- XML document classification [5], or ranking XML documents *w.r.t.* a set of DTDs [6, 7], [8],
- XML document and schema evolution [9, 10], [11], [12, 13], [14], [15], [16].

The main features of these proposals are presented in Section 6, and discussed in a contrastive study. Besides the existing proposals, considering the place now taken by XML in all information systems, it can be assumed that all the situations in which tree-to-language correction will be useful are not known yet.

This article is dedicated to a comprehensive presentation of an algorithm for correcting XML documents: principles, algorithms, proofs of properties and experimental results are provided. The presented algorithm is unique in its *completeness* in the sense that, given a non negative threshold th, the algorithm finds every tree t' valid with respect to S such that the edit distance between t and t' is no higher than th. As we show in our deep discussion of related work, this article is the first case of a full-fledged presentation of a solution in this important field, even if several proposals have been published during the last decade.

The resulting tool is available[3] under the GNU LGPL v3 license. This license allows one the use and the modification of the source codes, in order to adapt them to a particular application or to extend them so as to deal with XML Schema (XSD) following the guidelines that we provide in Section 4.4.

The paper is organized as follows: in Section 2 we review seminal results in the field of tree-to-tree edit distance and of word-to-language correction. Then we present a running example to illustrate our algorithm's principles. In Section 3 we introduce all the definitions that allow the reading of our algorithm, presented and analyzed in Section 4. We detail experimental results in Section 5. We end with the discussion of related work in Section 6 and a conclusion in Section 7.

[3]on the CODEX project webpage: `http://codex.saclay.inria.fr/deliverables.php`

2. BACKGROUND AND EXAMPLE

In this section we introduce the results underlying our XML document correction algorithm, and we provide some intuitions on its design via a basic example.

2.1. Seminal Results

Our generation of XML document corrections builds upon two fundamental algorithms. The first one, concerning trees, is Selkow's proposal for the tree-to-tree edit distance [17]. The second one, addressing strings, is Oflazer's computation of spelling corrections [18] based on a dynamic exploration of the finite state automaton that represents a dictionary.

The tree-to-tree editing problem addressed by [17] generalizes the problem of computing the edit distance between two strings [19] to the one of two unranked labeled trees. Three editing operations are considered: (i) changing a node label, (ii) deleting a subtree, (iii) inserting a subtree (the two latter operations can be decomposed into sequences of node deletions and insertions, respectively). A cost is assigned to each of these operations and the problem is to find the minimal cost of all operation sequences that transform a tree t into a tree t'. The edit distance between t and t' is equal to this minimal cost.

The computation of the edit distance is based on a matrix H where each cell $H[i,j]$ contains the edit distance between two partial trees $t\langle i\rangle$ and $t'\langle j\rangle$. A partial tree $t\langle i\rangle$ of a tree t consists of the root of t and its subtrees $t_{|_0}, \ldots, t_{|_{i-1}}$ – see Figure 1(a). We denote by $C_{i,j}$ the minimal cost of transforming $t\langle i\rangle$ into $t'\langle j\rangle$. Selkow has shown that $C_{i,j}$ is the minimum cost of three operation sequences: *(1)* transforming $t\langle i\rangle$ into $t'\langle j-1\rangle$ and inserting $t'_{|_j}$, *(2)* transforming $t\langle i-1\rangle$ into $t'\langle j-1\rangle$ and transforming $t_{|_i}$ into $t_{|_j}$, and *(3)* transforming $t\langle i-1\rangle$ into $t'\langle j\rangle$ and deleting $t_{|_i}$.

The matrix H is computed column by column, from left to right and top down. Thus, each element $H[i,j]$ is deduced from its three neighbors $H[i-1,j-1]$, $H[i-1,j]$ and $H[i,j-1]$, as shown in Figure 1(b). It contains the minimum value among *(1)* its left-hand neighbor's value plus the minimum cost of inserting the subtree $t'_{|_j}$ (Figure 1(b), edge (1)), *(2)* its upper-left-hand neighbor's value plus the minimum cost of transforming the subtree $t_{|_i}$ into $t'_{|_j}$ (Figure 1 (b), edge (2)), and *(3)* its upper neighbor's value plus the minimum cost of deleting the subtree $t_{|_i}$ (Figure 1 (b), edge (3)).

Example 1. Let t and t' be the two trees in Figure 2. Consider the cost of each elementary edit operation (inserting, deleting or renaming a node) equal to 1. The edit distance matrix H between t and t' is given in Figure 3. Each of its rows and columns is indexed by: (i) -1 when a tree's root is concerned, (ii) an integer i when the $(i+1)$-th child of a root is concerned. The row and column indices are accompanied by the labels of the corresponding nodes. The bottom right-hand cell of the matrix contains the edit distance between t and t', i.e. the cost of the minimal edit sequence transforming t into t'. This sequence consists of: relabeling the root to e, inserting b as the root's first child, and relabeling b (d's parent) to c. □

It should be noticed that computing the edit distance between t and t' implies computing edit distances between subtrees of t and subtrees of t'. The time complexity of Selkow's algorithm is $O(\Sigma_{i=0}^{min(d_t,d_{t'})} h_i h'_i)$, where d_t and $d_{t'}$ are the depths of t and t', and h_i and h'_i are the numbers of nodes at height i in t and t', respectively.

The computation of Selkow's tree edit distance $dist(t,t')$ is our first background, but we need more: our aim is to compute minimal operation sequences for transforming a tree t that is not valid with respect to a schema S into valid trees. For this purpose, we do not only compute a distance between the given tree t and the schema S, we actually compute *operation sequences* transforming t into trees that are valid with respect to S. Moreover, we do not limit the computation to minimal sequences, instead we search for all valid trees t' such that $dist(t,t') \leq th$, where th is a given distance threshold.

To this aim we follow the same ideas as in Oflazer's work [18], where an algorithm is presented that, for a given input string X not belonging to the language represented by a given finite state automaton (FSA) A, looks for all possible corrections (i.e. strings recognizable by A) whose distance from X is less than or equal to a distance threshold th.

This algorithm, although not addressed in the recent state-of-the-art report on the string-to-language correction by [20], can be classified – according to the taxonomy proposed in this report – as a direct method based on a prefix tree implemented as a string trie. More precisely, it is based on a dynamic exploration of the FSA representing the language. A partial candidate $Y = a_1a_2\ldots a_k$ is generated by concatenating labels of transitions, starting from the initial state q_0, until reaching a final state. Consider that we are in state q_m. In order to extend Y by the label b of an outgoing transition of q_m, it is checked whether the *cut-off edit distance* between X and the new word $Y = a_1a_2\ldots a_kb$ does not exceed th. The cut-off edit distance between X and Y is a measure introduced in [21] that allows one to cut the FSA exploration as soon as it becomes impossible that extending Y could reduce the edit distance between X and Y. If the cut-off edit distance exceeds th, then the last transition is canceled, the last character b is removed from the current candidate Y and the exploration goes on through other outgoing transitions of q_m. When a final state is reached during the generation of candidate Y, if $dist(X,Y) \leq th$, then Y is a valid candidate for correcting X. The following example, borrowed from [18], illustrates these ideas.

Example 2. Figure 4 shows the graph $G1$ representing

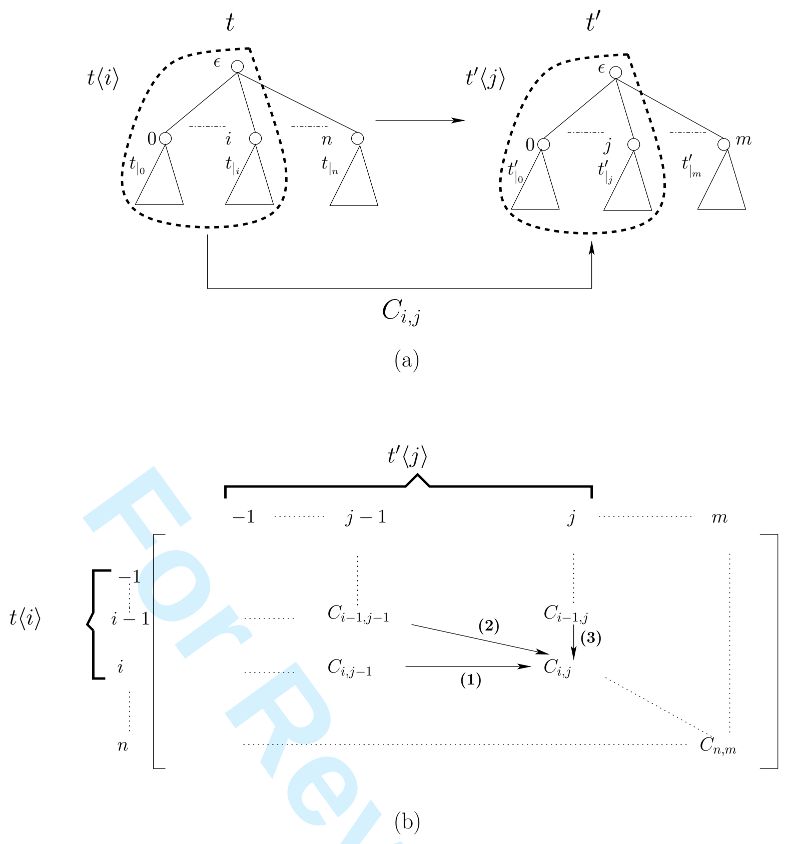


FIGURE 1. (a) Two partial trees $t\langle i\rangle$ and $t'\langle j\rangle$. (b) Tree edit distance matrix: computation of $H[i,j] = C_{i,j}$.

the finite-state automaton corresponding to the regular expression $E = (aba|bab)^*$ and, in $G2$, the exploration of this graph while considering the word $X = ababa$, which is not in the language $L(E)$. The three paths surrounded in $G2$ represent three correct words $abaaba$, $ababab$ and $bababa$. For each node n in $G2$, the brackets contain the cut-off value between the incorrect word $ababa$ and the word corresponding to the path connecting the initial state q_0 with the state in node n. If we consider a distance threshold $th = 1$, the three surrounded words are valid candidates. It can be noticed that no continuation of these three paths can lead to another candidate within the threshold $th = 1$ because the *cut-off* turns to 2 for all their following states. □

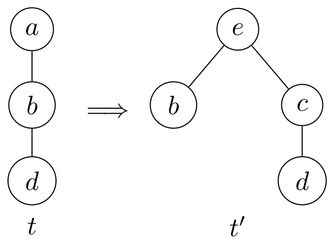


FIGURE 2. Compared trees t and t'.

In the same way as in [21], an edit distance matrix between X and each potential candidate Y is dynamically computed: $H[i,j]$ contains the distance between the prefixes of lengths i and j of the two strings

H		-1 e	0 b	1 c
-1	a	1	2	4
0	b	3	2	3

FIGURE 3. Edit distance matrix between t and t'.

X and Y. The added value of [18] is to make use of the finite-state representation of the lexicon so that, when a word is looked up in the lexicon, the initial columns in the matrix that correspond to the same common prefix of lexicon words are calculated only once.

To resume, our proposal directly builds on [17] and [18]. We admit Selkow's tree-to-tree edit distance based on three elementary operations (relabeling a node, inserting or deleting a leaf), and we use the dynamic programming method to calculate this distance via a distance matrix. However, we extend these ideas into correcting a tree with respect to a tree language similarly to how Oflazer extends a word-to-word distance calculation into correcting a word with respect to a word language.

In what follows, we introduce our proposal through an example: we show how we combine the two previous approaches in order to compute all tree edit operation sequences that transform a tree t into valid trees t' such that $dist(t,t') \leq th$.

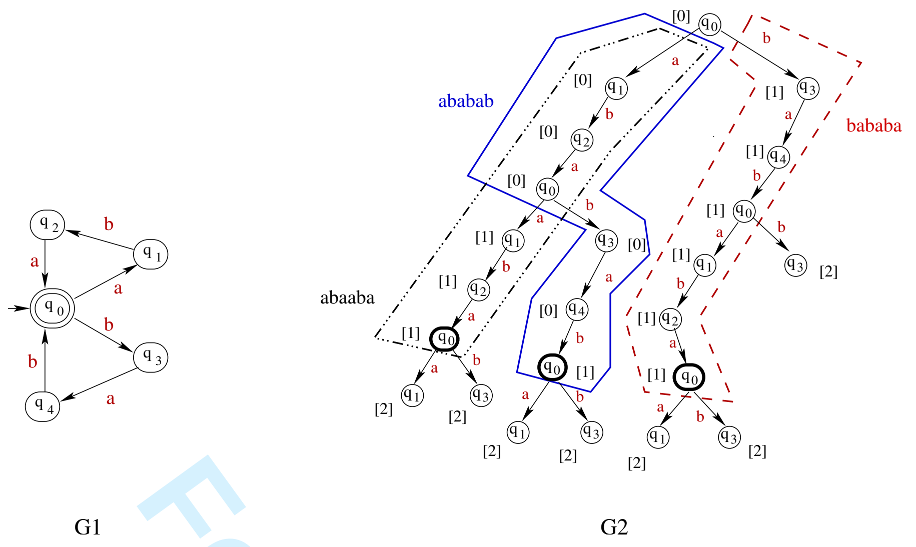


FIGURE 4. $G1$: graph representing the FSA that corresponds to $(aba|bab)^*$, $G2$: graph representing exploration paths for correcting the word $ababa$ with a threshold $th = 1$

2.2. Running Example

Let $\Sigma = \{root, a, b, c, d\}$ be a set of tags, and let t be the XML tree in Fig. 5. The positions of nodes in t are represented by sequences of integers such that: (i) the children of a node are numbered from left to right by consecutive non-negative integers 0, 1, etc., (ii) the tree's root is at position ϵ, (iii) if node n is at position p, the position of the $(i+1)$-th child of n is given by the concatenation of p and i. For instance, in Fig. 5, the node at position 1.0 (labeled with c) is the first child of the node at 1 (labeled b), which on its turn is the second child of the root at ϵ. As formally described in section 3.1, a tree is seen as a mapping from positions to labels. Thus, the tree in Fig. 5 can be described as the set $\{(\epsilon, root), (0, a), (0.0, c), (0.1, d), (1, b), \ldots\}$.

Let S be the structure description in Fig. 6 for an XML schema. Note in particular the finite-state automaton associated with the root element and corresponding to the regular expression $b^*|ab^*c$. The tree t is not valid *w.r.t.* S because the word which is formed by the tags of the children of the root node, i.e. abb, does not belong to $L(b^*|ab^*c)$.

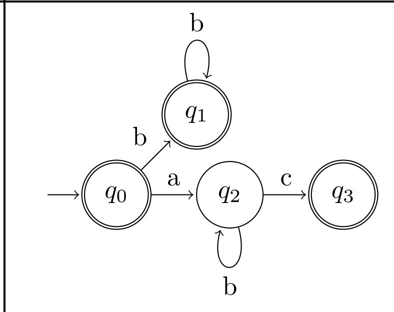
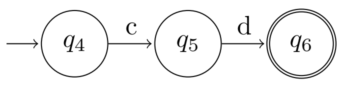
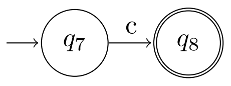
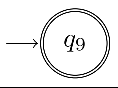
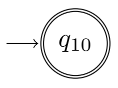
Tag	Regular Expression	Finite State Automaton(FSA)
root	$b^*\|ab^*c$	
a	cd	
b	c	
c	ϵ	
d	ϵ	

FIGURE 6. An example of a structure description.

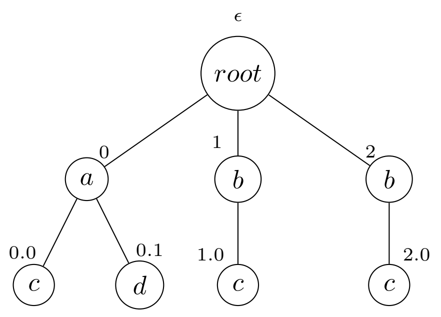


FIGURE 5. An XML tree.

We would like to compute the set of valid trees $\{t'_1, \cdots, t'_n\}$ whose distance from t is no higher than a given threshold th, for instance $th = 2$. Therefore, we perform a correction of t *w.r.t.* the schema S using a tree-to-language edit distance matrix M. This matrix contains the sets of operation sequences (of cost no higher than th each) needed to transform partial trees of t into partial trees of t'_i (we can have many possible corrections). We use $M[i][j]$ or (i, j) to indicate the

M		0 root	1 b	2 b	3 b	**4** ***b***
0	root	$\{nos_\emptyset\}$	$\{\langle(add,0,b),(add,0.0,c)\rangle\}$	$\emptyset$	$\emptyset$	$\boldsymbol{\emptyset}$
1	a	$\emptyset$	$\{os_1{=}\langle(relabel,0,b),(delete,0.1,/)\rangle\}$	$\emptyset$	$\emptyset$	$\boldsymbol{\emptyset}$
2	b	$\emptyset$	$\emptyset$	$\{os_1\}$	$\emptyset$	$\boldsymbol{\emptyset}$
3	b	$\emptyset$	$\emptyset$	$\emptyset$	$\{os_1\}$	$\boldsymbol{\emptyset}$

FIGURE 7. Content of the matrix M

cell of the matrix which is at line i and at column j. The first cell $(0,0)$ of the matrix contains the operation sequence needed to transform the root node of t to the root node of the trees in $L(S)$. Here, t has the same root node as the root node specified by the XML schema S so we keep this root intact. Thus, the first cell $(0,0)$ of the matrix M contains an empty operation sequence denoted by $nos_\emptyset$, as shown in Fig. 7. Then for computing the other cells of M we use the cells which are already computed. Namely, we concatenate each sequence taken from a cell above and/or to the left of the current cell with one of the three following, possibly complex, operations (provided that the threshold th is not exceeded):

(i) Inserting subtrees (denoted by $\rightarrow$): coming from the left-hand cell we concatenate its operation sequences with an insertion of a subtree in the result tree t'_i. Several different subtree insertions may be possible, which results in several sequences for each source sequence.

(ii) Correcting a subtree (denoted by $\searrow$): coming from the upper-left-hand cell we concatenate its operation sequences with a correction of a subtree of t into a valid subtree of t'_i. The correction of a subtree of t is performed by a recursive call so another tree-to-language edit distance matrix is computed.

(iii) Deleting a subtree (denoted by $\downarrow$): coming from the upper cell we concatenate its operation sequences with a deletion of a subtree in t.

In Fig. 7 going from cell $(0,0)$ to cell $(1,0)$ we consider deleting the subtree of t rooted at position 0, which has cost 3. Thus, the threshold is exceeded and cell $(1,0)$ becomes empty as well as all other cells below.

The computation of the matrix M is done column by column. A new column is added after following a transition in the FSA_{root} associated with the root element of S. For instance for the column $j=1$ we may use the transition (q_0,b,q_1) and this column will be referred to by the tag b. This means that the subtrees at position 0 in the correct tree that we are trying to construct will have a root labeled b. The tags for all columns $(0<j)$ in M form a word u. Fig. 7 shows the contents of the matrix M for the word $u = bbbb$. We explain now how we compute each internal cell of the column, for instance the cell $(1,1)$:

(i) We consider the left-hand cell $M[1][0]=\emptyset$, which is empty so it cannot yield any operation sequence.

(ii) We consider the upper-left-hand cell $M[0][0] = \{nos_\emptyset\}$ with cost equal to 0. We concatenate it with the operation sequence $os_1{=}\langle(relabel,0,b),(delete,0.1,/)\rangle$ which results from correcting the subtree $\{(\epsilon,a),(0,c),(1,d)\}$ at position 0 in t to a valid subtree with root b. The subtree that we obtain is $\{(\epsilon,b),(0,c)\}$. The cost of os_1 is $2 \le th = 2$ so we can add the resulting operation sequence set which contains os_1 itself to the cell $(1,1)$. The matrix which is computed for correcting the subtree $\{(\epsilon,a),(0,c),(1,d)\}$ into $\{(\epsilon,b),(0,c)\}$ is shown in Fig. 8. Note that os_1 stems from the sequence obtained here in cell $(2,1)$, prefixed with position 0.

M'		0 b	1 c
0	a	$\{\langle(relabel,\epsilon,b)\rangle\}$	$\{\langle(relabel,\epsilon,b),$ $(insert,0,c)\rangle\}$
1	c	$\{\langle(relabel,\epsilon,b),$ $(delete,0,/)\rangle\}$	$\{\langle(relabel,\epsilon,b)\rangle\}$
2	d	$\emptyset$	$\{\langle(relabel,\epsilon,b),$ $(delete,1,/)\rangle\}$

FIGURE 8. New matrix computed by a recursive call

(iii) We consider the upper cell
$M[0][1] = \{\langle(add,0,b),(add,0.0,c)\rangle\}$
with cost equal to 2. We concatenate this operation sequence with the operation sequence
$os_2 = \{\langle(delete,0.1,/),(delete,0.0,/),(delete,0,/)\rangle\}$
allowing us to delete the subtree at position 0 in t. However, the cost of the deletion of this subtree is 3 and its concatenation with $M[0][1]$ yields a sequence with cost 5, which exceeds the threshold 2. Thus we don't have, for the cell $(1,1)$, any operation sequence coming from the upper cell.

The computation of the cell (1,1), according to items (i),(ii),(iii) above, is illustrated in Fig. 9.

For the other cells of the matrix in Fig. 7, we use the transition (q_1,b,q_1). If the word formed by the column tags is in $L(FSA_{root})$ (i.e. we reach a final state), the bottom cell of the current column contains possible solutions. Since $bbb \in L(FSA_{root})$, cell $(3,3)$ contains an operation sequence capable of transforming t into a valid tree $t'_i \in L(S)$. When we apply this operation

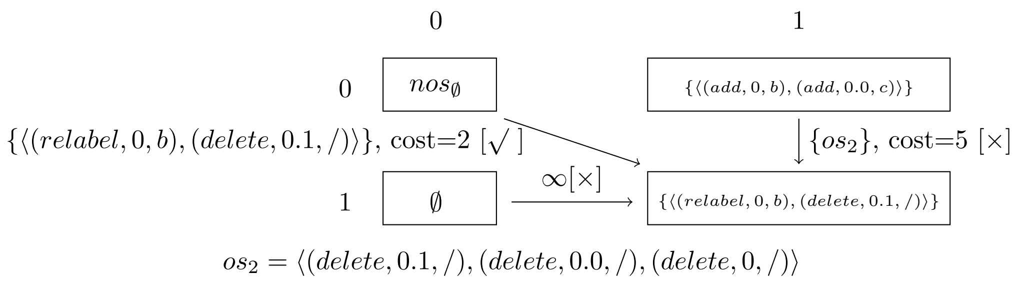

0
1
0
1
$nos_\emptyset$
$\{\langle(add,0,b),(add,0.0,c)\rangle\}$
$\{\langle(relabel,0,b),(delete,0.1,/)\rangle\}$, cost=2 [$\surd$]
$\{os_2\}$, cost=5 [$\times$]
$\emptyset$
$\infty[\times]$
$\{\langle(relabel,0,b),(delete,0.1,/)\rangle\}$
$os_2 = \langle(delete,0.1,/),(delete,0.0,/),(delete,0,/)\rangle$

FIGURE 9. Computation of the cell (1,1)

sequence, i.e. $os_1{=}\langle(relabel,0,b),(delete,0.1,/)\rangle$, on the tree t, we obtain the tree t'_1 in Fig. 10.

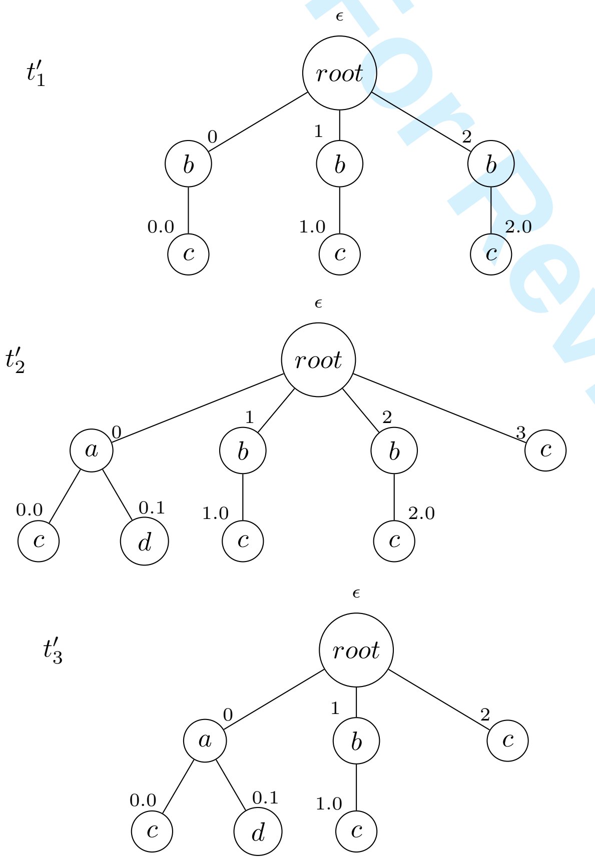


FIGURE 10. Three possible corrections t'_1, t'_2 and t'_3 for the tree t in Fig. 5

.

All the cells of the last column ($j = 4$) of the matrix in Fig. 7 are empty, which means that we can not have an operation sequence with a cost less than $th = 2$ for a word with the prefix *bbbb*. In this situation we backtrack by deleting the last column and try another transition. In this example we will delete all columns except the first one. After backtracking to q_0 it is possible to follow the transition (q_0, a, q_2) for computing the second column of the matrix in Fig. 11. The other columns of this matrix are computed by following the transition (q_2, b, q_2) until we reach another empty column. Note that the node operation sequence contained in cell $(3, 4)$ in Fig. 11 may be expressed as a single higher level operation on subtrees, namely as inserting a subtree $\{(\epsilon, b), (0, c)\}$ at position 3.

We backtrack again and use the transition (q_2, c, q_3). The cells of this current column (for $j = 4$) are shown in Fig. 12.

The word *abbc* formed by the tags of the current columns is in $L(FSA_{root})$ and the bottom cell of the current column contains a sequence with cost no higher than the threshold. Therefore we obtain a new correction t'_2 depicted in Fig. 10. In the state q_3 we don't have any outgoing transition so we backtrack, then we try the word *abc*. Fig. 13 shows the corresponding matrix, with a sequence in its bottom-right cell whose cost is not higher than th. This sequence is obtained with a new matrix computed by a recursive call in order to correct the subtree $\{(\epsilon, b), (0, c)\}$ at position 2 into $\{(\epsilon, c)\}$. The resulting correction t'_3 is depicted in Fig. 10. After that, we will have no more possibilities to find other corrections than t'_1, t'_2 and t'_3 within the threshold $th = 2$.

3. PRELIMINARY DEFINITIONS

In this section we provide formal definitions together with some intuitions concerning the notions and notations that are useful to present our algorithm and to discuss its properties in Section 4.

3.1. XML Trees and Tree Languages

We consider an XML document as an **ordered unranked labeled tree**, that we call an XML tree, defined as follows:

Definition 1. - **XML tree:** an XML tree is a mapping t from a set of **positions** $Pos(t)$ to an **alphabet** Σ, which represents the set of element names. For $v{\in}Pos(t)$, $t(v)$ is the label of the t's node at the position v. Positions are sequences of integers. As usual, ϵ denotes the empty sequence of integers, i.e. the **root position**. The character "."

M	0	1	2	3	4	**5**
	root	a	b	b	b	***b***
0 root	$\{nos_\emptyset\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\boldsymbol{\emptyset}$
1 a	$\emptyset$	$\{nos_\emptyset\}$	$\{\langle(add,1,b),$ $(add,1.0,c)\rangle\}$	$\emptyset$	$\emptyset$	$\boldsymbol{\emptyset}$
2 b	$\emptyset$	$\{\langle(delete,1.0,/),$ $(delete,1,/)\rangle\}$	$\{nos_\emptyset\}$	$\{\langle(add,2,b),$ $(add,2.0,c)\rangle\}$	$\emptyset$	$\boldsymbol{\emptyset}$
3 b	$\emptyset$	$\emptyset$	$\{\langle(delete,2.0,/),$ $(delete,2,/)\rangle\}$	$\{nos_\emptyset\}$	$\{\langle(add,3,b),$ $(add,3.0,c)\rangle\}$	$\boldsymbol{\emptyset}$

FIGURE 11. Content of the matrix M for $u = abbbb$ (after backtracking from state q_1 in FSA_{root})

M	0	1	2	3	**4**
	root	a	b	b	***c***
0 root	$\{nos_\emptyset\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\boldsymbol{\emptyset}$
1 a	$\emptyset$	$\{nos_\emptyset\}$	$\{\langle(add,1,b),$ $(add,1.0,c)\rangle\}$	$\emptyset$	$\boldsymbol{\emptyset}$
2 b	$\emptyset$	$\{\langle(delete,1.0,/),$ $(delete,1,/)\rangle\}$	$\{nos_\emptyset\}$	$\{\langle(add,2,b),$ $(add,2.0,c)\rangle\}$	$\emptyset$
3 b	$\emptyset$	$\emptyset$	$\{\langle(delete,2.0,/),$ $(delete,2,/)\rangle\}$	$\{nos_\emptyset\}$	$\boldsymbol{\{\langle(add,3,c)\rangle\}}$

FIGURE 12. Content of the matrix M after backtracking

M	0	1	2	**3**
	root	a	b	***c***
0 root	$\{nos_\emptyset\}$	$\emptyset$	$\emptyset$	$\boldsymbol{\emptyset}$
1 a	$\emptyset$	$\{nos_\emptyset\}$	$\{\langle(add,1,b),$ $(add,1.0,c)\rangle\}$	$\boldsymbol{\emptyset}$
2 b	$\emptyset$	$\{\langle(delete,1.0,/),$ $(delete,1,/)\rangle\}$	$\{nos_\emptyset\}$	$\boldsymbol{\{\langle(add,2,c)\rangle\}}$
3 b	$\emptyset$	$\emptyset$	$\{\langle(delete,2.0,/),$ $(delete,2,/)\rangle\}$	$\boldsymbol{\{\langle(relabel,2,c),}$ $\boldsymbol{(delete,2.0,/)\rangle\}}$

FIGURE 13. Content of the matrix M after the next backtracking

denotes the **concatenation** of sequences of integers. The set $Pos(t)$ is closed under prefixes[4] and for each position in $Pos(t)$ all its left siblings also belong to $Pos(t)$, which can be formally expressed as follows: $\forall_{i,j\in\mathbb{N}}\forall_{u\in\mathbb{N}^*}[[0{\leq}i{\leq}j, u.j{\in}Pos(t)] \Rightarrow u.i{\in}Pos(t)]$. The set of **leaves** of t is defined by:
$leaves(t) = \{u{\in}Pos(t) \mid \nexists_{i\in\mathbb{N}}\ u.i{\in}Pos(t)\}$.
We denote by $|t|$ the **size** of t, i.e. the number of positions in $Pos(t)$. We denote by $\bar{t}$ the **number of t's root's children**. We denote by $t_\emptyset$ an **empty tree** ($Pos(t_\emptyset) = \emptyset$).

EXAMPLE 3. Fig. 5 represents a sample XML tree t. We have:

- $\Sigma \supseteq \{root, a, b, c, d\}$
- $Pos(t) = \{\epsilon, 0, 0.0, 0.1, 1, 1.0, 2, 2.0\}$
- $t = \{(\epsilon, root), (0, a), (0.0, c), (0.1, d), (1, b), (1.0, c), (2, b), (2.0, c)\}$
- $t(\epsilon) = root, t(0) = a, t(0.0) = c$, etc.
- $leaves(t) = \{0.0, 0.1, 1.0, 2.0\}$
- $|t| = 8, \bar{t} = 3$

□

DEFINITION 2. **- Relationships on a Tree:** Let $p, q \in Pos(t)$. Position p is an **ancestor** of q and q is a **descendant** of p if q is a proper prefix of p, i.e. $p < q$ (cf. footnote).

EXAMPLE 4. In Fig. 5 positions ϵ and 2 are ancestors of position 2.0.

DEFINITION 3. **- Subtree and Partial Tree:** Given a non-empty XML tree t, a position $p \in \mathbb{N}^*$ and $i \in \mathbb{N}$ $s.t.$ $-1 \leq i \leq \bar{t} - 1$, we denote by:

- $t_{|_p}$, the **subtree** whose root is at position $p \in Pos(t)$, defined as follows:

1. Each node in t under p appears in $t_{|_p}$.

[4]The prefix relation in $\mathbb{N}^*$, denoted by $\leq$ is defined by: $u \leq v$ iff $u.w = v$ for some $w \in \mathbb{N}^*$. Sequence u is a proper prefix of v, i.e. $u < v$, if and only if $w \neq \epsilon$. A set $Pos(t) \subseteq \mathbb{N}^*$ is closed under prefixes if $u \leq v$, $v \in Pos(t)$ implies $u \in Pos(t)$.

Formally: $\forall_{u \in \mathbb{N}^*}[[p.u \in Pos(t)] \Rightarrow [u \in Pos(t_{|_p}) \text{ and } t_{|_p}(u) = t(p.u)]]$

2. Each node in $t_{|_p}$ appears in t under p. Formally: $\forall_{u \in Pos(t_{|_p})} p.u \in Pos(t)$

- $t\langle i \rangle$, the **partial tree** that contains the t's root and the subtrees rooted at the first $i+1$ children of t's root, defined as follows:
  1. Positions in $t\langle i \rangle$ are the same as in t's root and in its corresponding subtrees. Formally: $Pos(t\langle i \rangle) = \{v \in Pos(t) | v = \epsilon \text{ or } \exists_{0 \le k \le i} \exists_{u \in \mathbb{N}^*} v = k.u\}$
  2. Labels in $t\langle i \rangle$ are the same as in t. Formally: $\forall_{v \in Pos(t\langle i \rangle)} t\langle i \rangle(v) = t(v)$

Note that each subtree and each partial tree is a tree in the sense of Definition 1. Note also that for a given non empty tree t we have $t_{|_\epsilon} = t$, $t\langle \bar{t} - 1 \rangle = t$, and $t\langle -1 \rangle = \{(\epsilon, t(\epsilon))\}$. Given a tree t we denote by d_t the depth of t, i.e. the value resulting from applying the function ***depth(t)*** defined as follows:

1. $depth(t) = 0$ if $t = t_\emptyset$
2. $depth(t) = 1$ if $\exists_{l \in \Sigma} t = \{(\epsilon, l)\}$
3. $depth(t) = 1 + max_{i \in [0..\bar{t}-1]}\{depth(t_{|_i})\}$

Example 5. Fig. 14 shows the subtree $t_{|_1}$ and the partial tree $t\langle 1 \rangle$ related to the tree t in Fig. 5. We have: $depth(t_{|_1}) = 2$ and $depth(t\langle 1 \rangle) = 3$. □

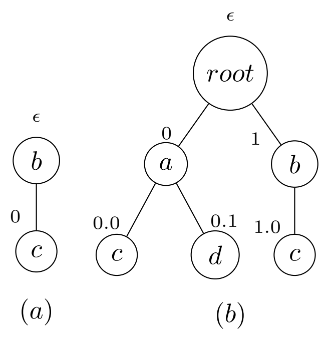


FIGURE 14. (a) The subtree $t_{|_1}$ and (b) the partial tree $t\langle 1 \rangle$ related to the tree t in Fig. 5

XML documents are seen in this paper as ordered unranked labeled trees that should respect some schema constraints expressed by a set of regular expressions that we call a *structure description*. We limit ourselves to elements, and to the DTD-equivalent case in which the content of each element name is defined by one and only one regular expression on Σ. We currently disregard constraints defined by a given DTD for XML attributes. We do not consider integrity constraints that might be expressed within a richer formalism such as XML Schema (XSD) either.

Definition 4. - **Structure Description:** A **structure description** S is a triple (Σ, *root*, *Rules*) where Σ is an alphabet (element names), *root* is the root label, and *Rules* is a set of pairs (a, FSA_a) such that $a \in \Sigma$ is a tag and FSA_a is the finite state automaton representing all possible words formed by the labels of the children of a node labeled by a. Formally:

1. $root \in \Sigma$
2. $Rules = \{(a, FSA_a) \mid a \in \Sigma\}$
3. $\forall_{a \in \Sigma}[FSA_a = (\Sigma_a, S_a, s_0^a, F_a, \Delta_a), \Sigma_a \subseteq \Sigma, s_0^a \in S_a, F_a \subseteq S_a, \Delta_a \subseteq S_a \times \Sigma_a \times S_a]$.

As usual, Σ_a, S_a, s_0^a, F_a and Δ_a are, respectively, the alphabet, the set of states, the initial state, the set of final states and the transition function of the finite-state automaton associated with a, respectively. Alternatively, we denote the transition function Δ_a by $FSA_a.\Delta$. The **word language** $L(FSA_a)$ defined by FSA_a is the set of all words accepted by FSA_a. We suppose that $\forall_{a \in \Sigma} L(FSA_a) \neq \emptyset$.

Example 6. The triple $S = (\Sigma, root, Rules)$ where $\Sigma = \{root, a, b, c, d\}$ and *Rules* are depicted in Figure 6 is a structure description. □

Definition 5. - **Locally Valid Tree:** Given a structure description $S = (\Sigma, root, Rules)$ a tree t is said to be **locally valid** with respect to S if and only if its labels belong to Σ, and it respects the constraints defined in *Rules*. Formally:

1. $\forall_{p \in Pos(t)} t(p) \in \Sigma$.
2. $\forall_{p \in Pos(t) \setminus leaves(t)} t(p.0)t(p.1) \ldots t(p.(\bar{t}_{|_p} - 1)) \in L(FSA_{t(p)})$, i.e. the labels of p's children form a word accepted by the automaton associated with p's label.
3. $\forall_{p \in leaves(t)} \epsilon \in L(FSA_{t(p)})$, i.e. the empty word is accepted by each automaton associated with a leaf.

Definition 6. - **Valid Tree:** Given a structure description $S = (\Sigma, root, Rules)$, a tree t is said to be **valid** with respect to S if and only if it is locally valid with respect to S and $t(\epsilon) = root$.

Definition 7. - **Tree Languages 1-2:** Given a structure description S, we introduce the following notations for the tree languages defined by S:

1. $L(S)$ denotes the set of all trees which are valid with respect to S.
2. $L_{loc}(S)$ denotes the set of all trees which are locally valid with respect to S.

Definition 8. - **Partially Valid Tree:** Given a structure description S, a tree t is said to be **partially valid** with respect to S if and only if it is a partial tree for a locally valid tree. Formally: $\exists_{t' \in L_{loc}(S)} \exists_{-1 \le i \le \bar{t'} - 1} t = t'\langle i \rangle$.

Example 7. The tree t in Fig. 5 is not valid with respect to the structure description S in Example 6. All subtrees

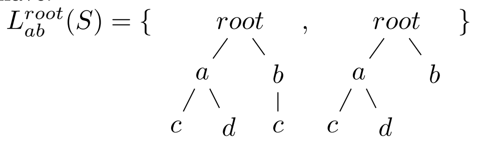
$t_{|0}, t_{|1}, t_{|2}$ are locally valid *w.r.t.* S. All partial trees $t\langle -1\rangle, t\langle 0\rangle, t\langle 1\rangle, t\langle 2\rangle$ are also partial trees of t'_2 in Fig. 10, thus they are partially valid trees. If the node $(0.0, c)$ is deleted in t then no partial tree of t is partially valid, except $t\langle -1\rangle$. □

DEFINITION 9. **- Tree Languages 3-4:** Given a structure description $S = (\Sigma, root, Rules)$, a label $c \in \Sigma$ and a word u being a prefix of a valid word $w \in FSA_c$, we introduce the following notations for the tree languages defined by S:

1. $L_{part}(S)$ denotes the set of all trees which are partially valid with respect to S.
2. $L^c_u(S)$ denotes the set of all trees which are partially valid with respect to S and have the word u under the root c. Formally: $L^c_u(S) = \{t \mid t \in L_{part}(S), t(\epsilon) = c \text{ and } t(0)\ldots t(\bar{t}-1) = u\}$.

Obviously, given a valid tree $t \in L(S)$, t is locally valid, all its subtrees are locally valid and all its partial trees are partially valid. As it is shown in Section 2.2 and detailed later on in this paper, correcting a tree t can be considered as dynamically building sets of partially valid trees close to t, extending the way Oflazer [18] dynamically builds words recognized by an FSA and satisfying the *cut-off* test with the word to be corrected (cf. Section 2.1).

EXAMPLE 8. Disregarding the tree positions, for the schema S in Example 6 modified in such a way that b is associated with the regular expression $c|\epsilon$ instead of c, we have:



□

Note that:

- if $u \in L(FSA_c)$ then $L^c_u \subseteq L_{loc}(S)$ *i.e.* all trees in L^c_u are locally valid;
- if $u \in L(FSA_{S.root})$ then $L^{S.root}_u \subseteq L(S)$ *i.e.* all trees in $L^{S.root}_u$ are valid;
- $\bigcup_{u \in L(FSA_{S.root})} L^{S.root}_u = L(S)$.

3.2. Operations on Trees

A tree may be changed through one or more node-edit operations, i.e. relabelings, insertions and deletions of nodes. Given a tree t, a *node-edit operation* may be applied to an *edit position* p provided that p respects some constraints depending on the type of the node-edit operation. For instance, an insertion of a new node in the tree in Fig. 5 is possible at any of its positions except ϵ but also at some still nonexistent positions, e.g. 2.1. A deletion of a node is possible at any leaf position. While inserting a node some positions may get shifted to the right, e.g. after an insertion at 2 position 2 becomes 3, 2.0 becomes 3.0, etc. While deleting a node some nodes get shifted to the left, e.g. after deleting 0.0 position 0.1 becomes 0.0, etc. Therefore, in order to define node-edit operations, we introduce the following sets of positions:

DEFINITION 10. **- Sets of Tree Positions:** Let t be a tree. Let p be a position such that $p \in Pos(t)$ and $p = \epsilon$ or $p = u.i$ (with $u \in \mathbb{N}^*$ and $i \in \mathbb{N}$). We define the following sets of positions in t:

- The **insertion frontier** of t is the set of all positions non existing in t on which it is possible to perform a node insertion. Formally:
  1. $InsFr(t_\emptyset) = \{\epsilon\}$.
  2. If $t \neq t_\emptyset$ then $InsFr(t) = \{v.j \notin Pos(t) \mid v \in Pos(t) \text{ and } j \in \mathbb{N} \text{ and } [(j = 0) \text{ or } ((j \neq 0) \text{ and } v.(j-1) \in Pos(t))]\}$.
- The **change position set** is the set of all positions that have to be either deleted or shifted left or right, in case of a node deletion or insertion at p. Formally:
  1. $ChangePos_\epsilon(t) = \{\epsilon\}$
  2. If $p \neq \epsilon$ then $ChangePos_p(t) = \{w \mid w \in Pos(t), w = u.k.u', i \leq k < \bar{t_u} \text{ and } u' \in \mathbb{N}^*\}$.
- The **shift-right position set** is the set of all target positions resulting from shifting a part of a tree as a result of inserting a new node at p. Formally:
  1. $ShiftRightPos_\epsilon(t) = \emptyset$.
  2. If $p \neq \epsilon$ then $ShiftRightPos_p(t) = \{w \mid w = u.(k+1).u',\ u.k.u' \in Pos(t),\ i \leq k < \bar{t_{|u}} \text{ and } u' \in \mathbb{N}^*\}$.
- The **shift-left position set** is the set of all target positions resulting from shifting a part of a tree as a result of deleting a node at p. Formally:
  1. $ShiftLeftPos_\epsilon(t) = \emptyset$.
  2. If $p \neq \epsilon$ then $ShiftLeftPos_p(t) = \{w \mid w = u.(k-1).u', u.k.u' \in Pos(t), i+1 \leq k < \bar{t_{|u}} \text{ and } u' \in \mathbb{N}^*\}$.

EXAMPLE 9. For the tree t in Fig. 5 we have:

- $InsFr(t) = \{0.0.0, 0.1.0, 0.2, 1.0.0, 1.1, 2.0.0, 2.1, 3\}$
- $ChangePos_1(t) = \{1, 1.0, 2, 2.0\}$
- $ShiftRightPos_1(t) = \{2, 2.0, 3, 3.0\}$
- $ShiftLeftPos_1(t) = \{1, 1.0\}$ □

DEFINITION 11. **- Node-Edit Operations:** Given an alphabet Σ and a special character $/ \notin \Sigma$ a **node-edit operation** ed is a tuple (op, p, l), where $op \in \{relabel, add, delete\}$, $p \in \mathbb{N}^*$ and $l \in \Sigma \cup \{/\}$. Given a tree t the node-edit operation ed is **defined on** t if and only if one of the following conditions holds:

- $op = relabel$, $l \in \Sigma$ and $p \in Pos(t)$
- $op = add$, $l \in \Sigma$ and $p \in Pos(t) \setminus \{\epsilon\} \cup InsFr(t)$
- $op = delete$, $l = /$ (empty label) and $p \in leaves(t)$

Given a node-edit operation ed we define an ***ed*-derivation** D_{ed} as a partial function on all trees on which ed is defined. An ed-derivation transforms a tree t into another tree t' (which is denoted by $t \xrightarrow{D_{ed}} t'$ or simply by $t \xrightarrow{ed} t'$) if the following holds:

- A **relabel** operation derivation replaces the label associated with the given position while leaving the rest of the tree intact. Formally, if $ed = (relabel, p, l)$ then:
  1. $Pos(t') = Pos(t)$,
  2. $t'(p) = l$,
  3. $\forall_{p' \in Pos(t') \setminus \{p\}} t'(p') = t(p')$.
- An **add** operation derivation inserts a single node at the given position while shifting some positions to the right and keeping all other positions intact. Formally, if $ed = (add, p, l)$ then:
  1. $Pos(t') = Pos(t) \setminus ChangePos_p(t) \cup ShiftRightPos_p(t) \cup \{p\}$,
  2. $t'(p) = l$,
  3. $\forall_{p' \in (Pos(t) \setminus ChangePos_p(t))} t'(p') = t(p')$,
  4. $\forall_{p' \in ShiftRightPos_p(t)}[[p = u.i,\ p' = u.(k+1).u'$ and $i, k \in \mathbb{N},\ u, u' \in \mathbb{N}^*] \Rightarrow t'(p') = t(u.k.u')]$.
- A **delete** operation derivation removes a leaf while shifting some positions to the left and keeping all other positions intact. Formally, if $ed = (delete, p, /)$ then:
  1. $Pos(t') = Pos(t) \setminus ChangePos_p(t) \cup ShiftLeftPos_p(t)$,
  2. $\forall_{p' \in (Pos(t) \setminus ChangePos_p(t))} t'(p') = t(p')$,
  3. $\forall_{p' \in ShiftLeftPos_p(t)}[[p = u.i,\ p' = u.(k-1).u'$ and $i, k \in \mathbb{N},\ u, u' \in \mathbb{N}^*] \Rightarrow t'(p') = t(u.k.u')]$.

Example 10. Consider the XML tree in Fig. 5 and the node-edit operation $ed = (add, 1, a)$ defined on t. The ed-derivation transforms t into t' shown in Fig. 15. □

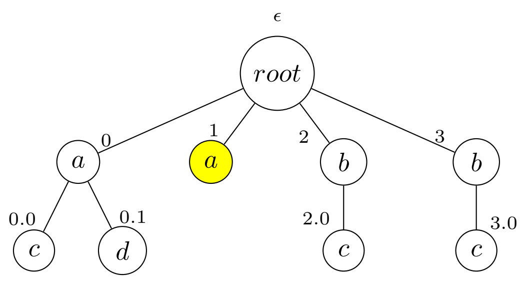


FIGURE 15. The result of the ed-derivation for $ed = (add, 1, a)$ over the tree t in Fig. 5.

Definition 12. - **Node-Edit Operation Sequence:** Let t be a tree. Let $0 \leq n$ and $ed_1, ed_2, \ldots ed_n$ be node-edit operations. The **node-edit operation sequence** $nos = \langle ed_1, ed_2, \ldots ed_n \rangle$ is **defined on** t if and only if there exists a sequence of trees $t_0, t_1, \ldots, t_n$ such that:

- $t_0 = t$
- $\forall_{0<k \leq n} ed_k$ is defined on t_{k-1} and $t_{k-1} \xrightarrow{ed_k} t_k$

Given a node-edit operation sequence nos we define a **nos-derivation** D_{nos} as a partial function on all trees on which nos is defined. A nos-derivation transforms a tree t into another tree t' (which is denoted by $t \xrightarrow{D_{nos}} t'$ or simply by $t \xrightarrow{nos} t'$) if and only if there exists a sequence of trees $t_0, t_1, \ldots, t_n$ defined as above and $t_n = t'$.
We denote by $nos_\emptyset$ the **empty sequence** of node-edit operations. The $nos_\emptyset$-derivation on a tree t leaves t intact, i.e. $t \xrightarrow{nos_\emptyset} t$.
Given two node-edit operation sequences nos_1 and nos_2 we say that nos_1 and nos_2 are **equivalent** if and only if for any tree t on which they are defined the nos_1-derivation and the nos_2-derivation on t lead to the same tree. Formally, $nos_1 \equiv nos_2$ if and only if:
$\forall_t[[nos_1$ and nos_2 are defined on t, $t \xrightarrow{nos_1} t_1$ and $t \xrightarrow{nos_2} t_2] \Rightarrow t_1 = t_2]$ □

Example 11. Let's consider the tree t in Fig. 5 and a node-edit operation sequence $nos = \langle (relabel, 0.1, c), (delete, 0.1, /), (relabel, 0, b) \rangle$. Clearly, nos is defined on t. The nos-derivation on t results in the tree t'_1 depicted in Fig. 10. Notice that we have:
$nos \equiv \langle (delete, 0.1, /), (relabel, 0, b) \rangle \equiv \langle (relabel, 0, b), (delete, 0.1, /) \rangle$. □

Let t be a tree and $NOS = \{nos_1, \ldots, nos_n\}$ be a set of node-edit operation sequences defined on t. For each $1 \leq i \leq n$ we can perform the nos_i-derivation on t in order to obtain a target tree t_i, i.e. $t \xrightarrow{nos_i} t_i$. We will denote this fact by $t \xrightarrow{NOS} \{t_1, \ldots, t_n\}$.

Some particular sequences of node-edit operations might be seen as higher-level operations where not only single nodes but whole subtrees intervene. For instance, the sequence of additions at positions 1, 1.0 and 1.1 in the tree in Fig. 5 may be seen as a single operation of inserting a 3-node subtree at position 1. Similarly, the sequence of deletions at positions 2.0 and 2 corresponds to removing the subtree rooted at position 2.

Definition 13. - **Tree-Edit Operations:** Given an alphabet Σ a **tree-edit operation** ted is a tuple (op, p, τ), where $op \in \{insert, remove\}$, $p \in \mathbb{N}^*$ and τ is a tree over Σ. Given a tree t the tree-edit operation ted is **defined on** t if and only if one of the following conditions holds:

- $op = insert$ and $p \in Pos(t) \setminus \{\epsilon\} \cup InsFr(t)$
- $op = remove$, $p \in Pos(t)$ and $\tau = t_\emptyset$

Given a tree-edit operation ted we define a **ted-derivation** D_{ted} as a partial function on all trees on which ted is defined. A ted-derivation transforms a tree t into another tree t' (which is denoted by $t \xrightarrow{D_{ted}} t'$ or simply by $t \xrightarrow{ted} t'$) if the following holds:

- An **insert** operation derivation inserts a new tree at the given position while shifting some positions to the right and keeping all other positions intact.

Formally, if $ted = (insert, p, \tau)$ then: $t = t_0 \xrightarrow{(add,p.v_1,\tau(v_1))} t_1 \xrightarrow{(add,p.v_2,\tau(v_2))} t_2 \cdots \xrightarrow{(add,p.v_n,\tau(v_n))} t_n = t'$ where $v_1, \ldots, v_n$ are the positions of τ reached in its prefix order traversal.

- A **remove** operation derivation removes a subtree rooted at the given position. Formally, if $ted = (remove, p, t_\emptyset)$ then: $t = t_0 \xrightarrow{(delete,p.v_1,/)} t_1 \xrightarrow{(delete,p.v_2,/)} t_2 \cdots \xrightarrow{(delete,p.v_n,/)} t_n = t'$ where $v_1, \ldots, v_n$ are the positions of $t_{|_p}$ reached in its inverted postfix (right-to-left) order traversal.

As shown in Definition 13, each tree-edit operation ted can be expressed in terms of a certain node-edit operation sequence nos. We will say in this case that ted and nos are **t-equivalent**, which is denoted by $ted \equiv_t nos$. Note that, by Definition 13, for each tree-edit operation there is exactly one t-equivalent node-edit operation.

EXAMPLE 12. Consider the tree t in Fig. 5 and the tree-edit operation $ted = (insert, 3, \tau_1)$ with τ_1=$\{(\epsilon, b), (0, c)\}$. Clearly, ted is defined on t. We have $ted \equiv_t \langle(add, 3, b), (add, 3.0, c)\rangle$. □

The fact of applying a node-edit operation (i.e. of performing the ed-derivation) induces a non-negative cost. In this paper the cost of each node-edit operation is fixed to one but that need not be the case in general[5].

DEFINITION 14. **- Operation Sequence Cost:** For any node-edit operation ed, we define $cost(ed)$ to be the non-negative cost of performing the ed-derivation on a tree. Given a node-edit operation sequence $nos = \langle ed_1, ed_2, \ldots, ed_n\rangle$ the cost of nos is defined as $Cost(nos) = \Sigma_{i=1}^{n}(cost(ed_i))$. The cost of a tree-edit operation ted is equal to the cost of the node-edit operation sequence nos which is t-equivalent to ted, i.e. $Cost(ted) = Cost(nos)$ for $ted \equiv_t nos$. Given a set of node-edit operation sequences NOS, we define the minimum cost on NOS as follows: $MinCost(NOS) = \min_{nos \in NOS} \{Cost(nos)\}$.

3.3. Operators on Sets of Operation Sequences

We now introduce some operators on sets of node-edit operation sequences that will allow an easy manipulation of these sequences in both the correction algorithm and its analysis.

DEFINITION 15. **- Minimum-Cost Subset:** Let NOS be a set of node-edit operation sequences. We denote by $MinCostSubset(NOS)$ the **minimum-cost subset** of NOS defined as the set of all sequences in NOS having no equivalent sequences in NOS with lower costs. Formally: $MinCostSubset(NOS) = \{nos \mid nos \in NOS$ and $\nexists_{nos' \in NOS}[nos' \equiv nos$ and $Cost(nos') < Cost(nos)]\}$.

DEFINITION 16. **- Minimum-Cost Union:** Let NOS_1 and NOS_2 be two sets of node-edit operation sequences. We denote by $NOS_1 \Cup NOS_2$ the **minimum-cost union** of NOS_1 and NOS_2 defined as follows: $NOS_1 \Cup NOS_2 = MinCostSubset(NOS_1 \cup NOS_2)$.

Let NOS_1 and NOS_2 be two sets of node-edit operation sequences. We denote by $NOS_1.NOS_2$ the concatenation of NOS_1 and NOS_2 such that $NOS_1.NOS_2 = \{nos_1.nos_2 \mid nos_1 \in NOS_1, nos_2 \in NOS_2\}$.

EXAMPLE 13. Let $S_1 = \{\langle(add, 1, c), (delete, 2, /), (relabel, 0, d)\rangle, \langle(relabel, 2, c), (delete, 2.0, /)\rangle\}$, and $S_2 = \{\langle(relabel, 0, a)\rangle\}$.
We have $S_1.S_2 = \{\langle(add, 1, c), (delete, 2, /), (relabel, 0, d), (relabel, 0, a)\rangle, \langle(relabel, 2, c), (delete, 2.0, /), (relabel, 0, a)\rangle\}$.
□

Notice that if either NOS_1 or NOS_2 is the empty set $\emptyset$ then $NOS_1.NOS_2 = \emptyset$.

DEFINITION 17. **- Threshold-Bound Concatenation:** Let NOS_1 and NOS_2 be two sets of node-edit operation sequences. Let th be a threshold ($th \geq 0$). We define the **threshold-bound concatenation** of NOS_1 and NOS_2, denoted by $NOS_1 ._{th} NOS_2$, as the subset of the concatenation $NOS_1.NOS_2$ in which all sequences have costs no greater than th. Formally: $NOS_1 ._{th} NOS_2 = \{nos_1.nos_2 \mid nos_1 \in NOS_1, nos_2 \in NOS_2, Cost(nos_1.nos_2) \leq th\}$. We extend the notion of the threshold-bound concatenation to sets of tree-edit operations. Namely, let TED_1 and TED_2 be sets of tree-edit operations and NOS be a set of node-edit operation sequences. Let NOS_{TED_i} (with $i \in \{1, 2\}$) be the set of node-edit operation sequences which are t-equivalent to the tree-edit operations in TED_i, i.e. $NOS_{TED_i} = \{nos \mid \exists_{ted \in TED_i}\ ted \equiv_t nos\}$. Then we assume the following definitions:

- $TED_i ._{th} NOS = NOS_{TED_i} ._{th} NOS$,
- $NOS ._{th} TED_i = NOS ._{th} NOS_{TED_i}$,
- $TED_1 ._{th} TED_2 = NOS_{TED_1} ._{th} NOS_{TED_2}$.

EXAMPLE 14. Let $\tau_1 = \{(\epsilon, a), (0, b), (1, c)\}$,
$\tau_2 = \{(\epsilon, e), (0, f)\}$, $\tau_3 = \{(\epsilon, g), (0, h)\}$,
$TED_1 = \{(insert, 1, \tau_1), (insert, 3, \tau_2)\}$,
$TED_2 = \{(insert, 0, \tau_3)\}$,
$NOS = \{\langle(relabel, \epsilon, root), (add, 2, b)\rangle, \langle(delete, 4, /)\rangle, nos_\emptyset\}$.
We have:
$NOS_{TED_1} = \{\langle(add, 1, a), (add, 1.0, b), (add, 1.1, c)\rangle,$
$\langle(add, 3, e), (add, 3.0, f)\rangle\}$
$NOS_{TED_2} = \{\langle(add, 0, g), (add, 0.0, h)\rangle\}$
$TED_1 ._3 NOS = \{\langle(add, 1, a), (add, 1.0, b), (add, 1.1, c)\rangle,$
$\langle(add, 3, e), (add, 3.0, f), (delete, 4, /)\rangle,$
$\langle(add, 3, e), (add, 3.0, f)\rangle\}$
$TED_2 ._3 NOS = \{\langle(add, 0, g), (add, 0.0, h), (delete, 4, /)\rangle,$
$\langle(add, 0, g), (add, 0.0, h)\rangle\}$
$TED_1 ._3 TED_2 = \emptyset$. □

[5]In our tool, operation costs are parameters of the correction process.

Definition 18. **- Prefixed Operation Sequence Set:** Let NOS be a set of node-edit operation sequences, and $u \in \mathbb{N}^*$. We define the **prefixed operation sequence set**, denoted by $AddPrefix(NOS, u)$, as the set resulting from adding the prefix u to all positions of the node-edit operations in NOS. Formally: $AddPrefix(NOS, u) = \{\langle ed_1, ed_2, \ldots, ed_n\rangle \mid ed_i = (op_i, u.pos_i, l_i)$ for $1 \leq i \leq n$ and $\exists_{\langle ed'_1, ed'_2, \ldots, ed'_n\rangle \in NOS}\ ed'_i = (op_i, pos_i, l_i)\}$

3.4. Distances and Corrections

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

Definition 19. **- Tree Distances:** Let t and t' be trees. Let $NOS_{t \to t'}$ be the set of all node-edit operation sequences nos such that $t \xrightarrow{nos} t'$. The distance between t and t' is defined by: $dist(t, t') = MinCost(NOS_{t \to t'})$. The distance between a tree t and a tree language L is defined by: $DIST(t, L) = \min_{t' \in L}\{dist(t, t')\}$.

Note that introducing the straightforward correspondence between node-edit and tree-edit operation sequences in Definition 13 highlights the equivalence between our tree distance definition and Selkow's one [17].

Example 15. Let us consider the tree t in Figure 5 and the schema S in Example 6. We have $DIST(t, L) = dist(t, t'_2) = Cost(\langle(add, 3, c)\rangle) = 1$, with t'_2 in Fig. 10. □

Definition 20. **- Tree Correction Set:** Given a tree t, a structure description $S = (\Sigma, root, Rules)$ and a threshold th ($th \geq 0$) we define the **correction set** of t with respect to S under th as the set of all valid trees whose distance from t is no greater than th. Formally: $L_t^{th}(S) = \{t' \mid t' \in L(S), dist(t, t') \leq th\}$.

The aim of the algorithm presented in Section 4 is to show how to obtain the correction set of the given tree t. More precisely, the algorithm provides the set of all node-edit operation sequences allowing us to obtain a tree t' belonging to t's correction set. Each of such node-edit operation sequences can be expressed in terms of operations equivalent to those defined by Selkow [17] (node relabeling, subtree insertion and subtree deletion). The conversion between node-edit and tree-edit operation sequences is straightforwardly deducible from Definition 13.

Note that with Definition 9 we have: $L_t^{th}(S) = \{t' \mid t' \in \bigcup_{u \in L(FSA_{S.root})} L_u^{S.root}(S)$ and $dist(t, t') \leq th\}$.

4. ALGORITHM

Having introduced all necessary definitions in the previous section, we are now going to provide a formal presentation of our algorithm and prove its properties, i.e. its completeness, soundness and termination, as well as its time complexity. The section ends with a discussion on several direct extensions.

Consider a schema $S = (\Sigma, root, Rules)$, a tree t and a natural threshold th. For correcting a tree t with respect to S under the threshold th, we use a dynamic programming method which calculates a two dimensional tree-to-language edit distance matrix M_u^c where c is a tag and $u = u_1 u_2 \ldots u_k$ is a word (sequence of tags). Each cell of M_u^c contains a set of node-edit operation sequences. Namely, $M_u^c[i][j]$ contains the set of all node-edit operation sequences transforming the partial tree $t\langle i{-}1\rangle$ into trees $t_1, \ldots, t_n$ each of which:

- is partially valid;
- has the root c and its root's children form a prefix[6] of u of length j;
- its distance from t is no greater than th.

Formally:
$\exists_{t_1, \ldots, t_n}[t\langle i{-}1\rangle \xrightarrow{M_u^c[i][j]} \{t_1, \ldots, t_n\}$ and $\forall_{1 \leq k \leq n}[t_k \in L^c_{u[1\ldots j]}(S)$ and $dist(t, t_k) \leq th]]$.

With $c = root$, $i = \bar{t}$ and $j = |u|$ the cell $M_u^c[i][j]$ contains the set of operation sequences capable of transforming a tree t into a set of trees belonging to $L_t^{th}(S)$.

4.1. Presentation

Matrix M_u^c can be computed by the function *correction* presented below. It takes as input parameters the tree t to be corrected, the structure description S, the threshold th and the root label intended for t. It returns the set of all node-edit operation sequences that transform t into locally valid trees with root c. When called with $c = root$ the function returns the set of all operation sequences capable of transforming a tree t into the whole correction set $L_t^{th}(S)$.

The first instance of the function *correction* will usually imply other instances whose results are all collected in the set *Result* that is returned at the end.

If the threshold is null while the initial tree t is locally valid and has the intended root c then the correction result is the empty sequence of operations (lines 2–3) since no operation needs to be applied to t. If however t is not locally valid no solution is possible with a non positive threshold (lines 5–6), thus the set of solutions is empty (which is different from the empty sequence being the only solution). With a positive threshold the matrix M_u^c is initialized with one column corresponding to $u = \epsilon$ and as many rows as the number of the root's subtrees plus one since row 0 corresponds to the root (lines 8–10). Then the cells of this first column are calculated (lines 11–19). Namely, the cell $M_u^c[0][0]$ receives the operation necessary to introduce the correct

[6] For $u = u_1 u_2 \ldots u_j \ldots u_n \in \Sigma^*$ we denote by $|u|$ the length of u, i.e. $|u| = n$, and by $u[1..j]$ the u's prefix of length j, i.e. $u[1..j] = u_1 u_2 \ldots u_j$ with $1 \leq j \leq n$.

```
Function correction($t$, $S$, $th$, $c$) return $Result$
Input
$t$: XML tree (to be corrected)
$S$: structure description
$th$: natural (threshold)
$c$: character (intended root tag of resulting trees)
Output
$Result$: set of node-edit operation sequences (allowing us to get resulting trees)
1.  begin
2.    if $th = 0$ and $t \in L_{loc}(S)$ and $t(\epsilon) = c$ then
3.      return $\{nos_\emptyset\}$ //Stop recursion
4.    else
5.      if $th \leq 0$ then
6.        return $\emptyset$ //Stop recursion
7.      else
8.        $u := \epsilon$
9.        $n := \bar{t}$ //$n$ is the number of t's root's children
10.       $M_u^c := newMatrix(n{+}1, 1)$ //Initialize the matrix with $n + 1$ rows and 1 column
          //Compute the first column in the matrix.
11.       if $t = t_\emptyset$ then
12.         $M_u^c[0][0] := \{(add, \epsilon, c)\}$
13.       else
14.         if $c = t(\epsilon)$ then
15.           $M_u^c[0][0] := \{nos_\emptyset\}$
16.         else
17.           $M_u^c[0][0] := \{(relabel, \epsilon, c)\}$
18.       for $i := 1$ to $n$ do
19.         $M_u^c[i][0] := \{(remove, i{-}1, t_\emptyset)\}._{th} M_u^c[i{-}1][0]$
20.       $Result := \emptyset$
          //This call to $correctionState$ begins the correction of $t$'s root's children
21.       $correctionState(t, S, th, c, M_u^c, initialState(FSA_c), Result)$
          //Function initialState returns the initial state of the FSA associated with $c$
22.       return $Result$
23. end
```

```
Procedure correctionState($t$, $S$, $th$, $c$, $M_u^c$, $s$, $Result$)
Input
$t$: XML tree (to be corrected)
$S$: structure description
$th$: natural (threshold)
$c$: character (intended root tag)
$M_u^c$: current matrix
$s$: current state in $FSA_c$
Input/Output
$Result$: set of node-edit operation sequences, which may be completed with corrections induced by the state $s$
1. begin
     //Check if $u \in L(FSA_c)$. If so add the bottom right hand side matrix cell to the result.
2.   if $s \in FSA_c.F$ then
3.     $Result := Result \Cup M_u^c[\bar{t}][|u|]$
4.   for all $\delta \in FSA_c.\Delta$ such that $\delta = (s, a, s')$ do
5.     $correctionTransition(t, S, th, c, M_u^c, s', a, Result)$
6. end
```

root c, i.e. (i) the addition of c if t is empty (lines 11–12), (ii) the empty sequence if the root is correct (lines 14–15), (iii) the relabeling operation if the root is incorrect (line 17). Notice that if t is the empty tree $t_\emptyset$ then the matrix M_u^c has only one line thus only this first cell is computed.

All cells below $M_u^c[0][0]$ are to represent the operation sequences transforming partial trees $t\langle i{-}1\rangle$ into a tree having only the root c. Therefore, these sequences contain the previously calculated operation for correcting the root, concatenated with deletions of all subtrees of the root (line 19). Note that these deletions: (i) are performed from right to left in order to save position shifting, (ii) are expressed –

for the sake of simplicity and complexity saving – by tree-edit operations (*remove*) reduced into node-edit operation sequences while performing the threshold-bound concatenation $\cdot_{th}$. Finally, the matrix M^c_u with the first initialized column is passed to the function *correctionState* (line 21), whose result becomes the result of the whole correction process (line 22).

Procedure *correctionState* performs the depth-first search exploration of the automaton FSA_c associated with the root c of the input tree. If the input state is final then the word u read until now while traversing FSA_c is valid with respect to FSA_c. Thus, all solutions accumulated in the bottom right-hand side cell of the current matrix M^c_u lead to partially valid trees and can be added to the set of solutions (lines 2–3). Then each transition δ outgoing from the current state s is considered (lines 4–5).

Procedure *correctionTransition* treats the current transition δ with its label a in order to correct partial trees of t into partially valid trees having a as the root of their last subtrees. This treatment consists in (i) computing one column, (ii) verifying whether this exploration path can go on, and if so, (iii) going on following this path by calling again the procedure *correctionState*. It can be noticed from the procedures *correctionState* and *correctionTransition* that the number of columns computed for the thread of the correction of the tree t w.r.t. the label c is bounded by $f_c^{\bar{t}+th}$ where f_c is the maximum fan-out of all states in FSA_c. This can be verified in the example detailed in Section 2.2.

Word v is formed by the labels read until now while traversing FSA_c, including the current transition's label a (line 3). The matrix M^c_v is initialized with as many columns as v's length plus 1, and as many rows as the number of the root's subtrees plus one (lines 4–6). The whole contents of M^c_v is recopied from the preceding matrix M^c_u (line 7) except the last column corresponding to the current transition, which is computed in lines 8–20. Namely, we first compute all node-edit operation sequences allowing us to transform an empty tree $t_\emptyset$ into a locally valid tree with root a (lines 8 and 13). Note that these sequences are t-equivalent to the tree-edit operations of inserting new locally valid trees at position $m-1$ in t. Any of these operations may potentially intervene only after having corrected the partial tree $t\langle i-1\rangle$ (i.e. for $i = 0$: only a root) into a partially valid tree $t' \in L^c_u$. Thus, the threshold allowed for inserting a new tree with root a at $m-1$ cannot exceed the general threshold th reduced by the cost of the previous least costly correction of $t\langle i-1\rangle$ into a t' ($th - MinCost(M^c_v[i][m-1])$). As this value varies, this call to *correction* is performed for each cell of the column.

The first cell of column m in the matrix corresponds to transforming the partial tree $t\langle -1\rangle$ into any of the trees $t'' \in L^c_v$. Thus, its contents is formed by previous corrections of $t\langle -1\rangle$ into a $t' \in L^c_u$ combined with the subtree insertions at $m-1$ prefixed by the insertion position $m-1$ (line 9). Each time operation sequences are combined, the threshold-bound concatenation $\cdot_{th}$ specified in Definition 17 is applied, in order to keep only those resulting sequences which do not exceed the threshold.

All other cells in column m are built by taking into account the three possibilities issued from Selkow's proposal (cf. Section 2):

- First we transform the partial tree $t\langle i-1\rangle$ into a $t' \in L^c_u$, then we insert a new locally valid subtree with root a at position $m-1$ (line 15). This corresponds to the horizontal correction possibility in the matrix shown in Fig. 1(b) and in Fig. 9.
- First we transform the partial tree $t\langle i-2\rangle$ into a $t' \in L^c_u$, then we transform the subtree $t_{|i-1}$ into a locally valid subtree with root a (line 16). This corresponds to the diagonal correction possibility in the matrix shown in Fig. 1(b) and in Fig. 9.
- First we remove the subtree $t_{|i-1}$, then we transform the partial tree $t\langle i-2\rangle$ into a $t' \in L^c_v$ (line 17). This corresponds to the vertical correction possibility in the matrix shown in Fig. 1(b) and in Fig. 9.

All sequences induced by these three possibilities are stored provided that: (i) they do not exceed the threshold (this verification is performed by the threshold-bound concatenation $\cdot_{th}$), (ii) they have no equivalent sequences with lower cost (which is guaranteed by the the minimum-cost union $\Cup$). Note that, here again, the tree-edit operations (tree insertions and deletions) are never explicitly stored. They are replaced instead by the t-equivalent node-edit operation sequences as a result of a recursive correction (line 8) and of the threshold-bound concatenation (line 15). Some cells of the current column may contain an empty set after the application of the threshold-bound concatenation $\cdot_{th}$. Variable *nbSolInColumn* counts the number of cells in the current column m which contain at least one solution (lines 2, 10–11 and 18–19). If the current column contains at least one solution then the recursive correction goes on with the arrival state of the current transition (line 22). Otherwise this exploration path is cut off and a backtracking from the current transition is performed (i.e. the current matrix with the newly computed column will no more be used).

It is important to notice that the result of $correction(t, S, th, c)$ is exactly the following set of node edit operation sequences:
$\{nos \mid nos \in \bigcup_{u \in L(FSA_c)} M^c_u[\bar{t}][|u|] \text{ and } Cost(nos) \leq th\}$
That is because lines 4–5 of procedure *correctionState* try all possible words in $L(FSA_c)$ and line 2 adds to the result only the operation sequences of the matrix $M^c_u[n][|u|]$ for which $u \in L(FSA_c)$. Thus, only those cells $M^c_u[\bar{t}][|u|]$ are selected which correspond

```
Procedure correctionTransition($t$, $S$, $th$, $c$, $M_u^c$, $s$, $a$, $Result$)
Input
$t$: XML tree (to be corrected)
$S$: structure description
$th$: natural (threshold)
$c$: character (intended root tag)
$M_u^c$: current matrix
$s$: target state of the current transition in $FSA_c$
$a$: label of the current transition
Input/Output
$Result$: set of node-edit operation sequences (with corrections induced by previously visited states)
1.  begin
2.    $nbSolInColumn := 0$
3.    $v := u.a$
4.    $m := |v|$ //$m$ is the length of the current word
5.    $n := \bar{t}$ //$n$ is the number of t's root's children
6.    $M_v^c := newMatrix(n+1, m+1)$ //Initialize a new matrix with $n$+1 rows and $m$+1 columns
7.    $M_v^c[0..n][0..m-1] := M_u^c[0..n][0..m-1]$ // Copy all the columns of $M_u^c$ into $M_v^c$
8.    $T := correction(t_\emptyset, S, th - MinCost(M_v^c[0][m-1]), a)$ //Compute the last column of $M_v^c$
9.    $M_v^c[0][m] := M_v^c[0][m-1].{}_{th}AddPrefix(T, m-1)$
10.   if $M_v^c[0][m] \neq \emptyset$ then
11.     $nbSolInColumn := nbSolInColumn + 1$
12.   for $i := 1$ to $n$ do
13.     $T := correction(t_\emptyset, S, th - MinCost(M_v^c[i][m-1]), a)$
14.     $M_v^c[i][m] :=$
15.       $M_v^c[i][m-1].{}_{th}AddPrefix(T, m-1)$ $\uplus$ //Horizontal correction
16.       $M_v^c[i-1][m-1].{}_{th}$ $correction(t_{|_{i-1}}, S, th - MinCost(M_v^c[i-1][m-1]), a)$ $\uplus$//Diagonal correction
17.       $\{(remove, i-1, t_\emptyset)\}.{}_{th}M_v^c[i-1][m]$ //Vertical correction
18.     if $M_v^c[i][m] \neq \emptyset$ then
19.        $nbSolInColumn := nbSolInColumn + 1$
20.   end for
      //If the last column of $M_v^c$ contains at least one cell with a cost less than $th$ the iteration goes on by calling correctionState
      //with the state s. Otherwise the iteration stops.
21.   if $nbSolInColumn \geq 1$ then
22.      $correctionState(t, S, th, c, M_v^c, s, Result)$
23. end
```

to the operation sequences leading to valid trees. Moreover, the threshold-bound concatenation operator $(\cdot_{th})$ together with the minimum-cost union operator $(\uplus)$ allow only non redundant operation sequences whose cost is less than or equal to th to be kept in M_u^c's cells.

EXAMPLE 16. It can be verified that the corrections found for the example given in Section 2.2 are precisely those computed by the function *correction*.

4.2. Properties

Let's focus initially on correcting an empty tree *w.r.t.* a schema S. We claim that the function *correction* finds out how to transform an empty tree into all locally valid trees within the threshold. This can be formally expressed by the following lemma.

LEMMA 1. Given a tag $c \in \Sigma$, a schema S and a threshold th, let S' be the schema $(\Sigma, c, S.Rules)$. A call to correction($t_\emptyset$,S,th,c) always terminates and returns the set $Result$ *s.t.* the following proposition holds: $t_\emptyset \xrightarrow{Result} L^{th}_{t_\emptyset}(S')$.

Proof. The proof is done by induction on the threshold th.

- **Basis**: If $th = 0$ then the function $correction(t_\emptyset, S, 0, c)$ returns in line 6 (an empty tree is not locally valid) with $Result = \emptyset$. Note that $t_\emptyset \xrightarrow{\emptyset} \emptyset$. Note also that $L^0_{t_\emptyset}(S') = \{t' \mid t' \in L(S'), dist(t_\emptyset, t') = 0\} = \emptyset$ since each t' in $L^0_{t_\emptyset}(S')$ must be equal to $t_\emptyset$ and $t_\emptyset$ is never valid *w.r.t.* a schema (it has no root). We can conclude that $t_\emptyset \xrightarrow{\emptyset} L^0_{t_\emptyset}(S')$.

- **Induction step**: Suppose that $0 < th$ and for each $0 \leq th' < th$ the call $correction(t_\emptyset, S, th', c)$ terminates and the proposition holds for its result. We will show that $correction(t_\emptyset, S, th, c)$ also terminates and the proposition holds for its result.

(a) <u>*Termination and soundness*</u>: We wish to show that $correction(t_\emptyset, S, th, c)$ terminates and each node operation sequence in $Result$ leads to a tree in

$L^{th}_{t_\emptyset}(S')$, i.e. a locally valid tree with root c and whose distance from $t_\emptyset$ is no greater than th.

Note that for $0 < th$ the call $correction(t_\emptyset, S, th, c)$ constructs a matrix M^c_v where $M^c_v[0][0] = \{(add, \epsilon, c)\}$ and $cost((add, \epsilon, c)) = 1$ (see function *correction*, line 12). The calculation of this first cell is straightforward so it clearly terminates. Then, for each $0 < j \leq |v|$, $M^c_v[0][j]$ is obtained by concatenating sequences from $M^c_v[0][j-1]$ with results of a new correction of $t_\emptyset$ *w.r.t.* S (see procedure $correctionTransition$, lines 8–9) with $th'' = th - MinCost(M^c_v[0][j-1])$. Since the correction of an empty tree induces at least one node insertion we have $th'' < th$. By hypothesis, this new correction terminates and yields sequences leading to locally valid trees. Finally, lines 2–3 in $correctionState$ guarantee that each final concatenation (if any) added to $Result$ leads to a tree t' which has a root c, and whose root's children form a word valid *w.r.t.* FSA_c. Thus, t' is necessarily locally valid, which proves soundness. Note also that the number of columns in M^c_v cannot exceed th. Thus, the algorithm terminates after at most th recursive calls.

(*b*) <u>Completeness</u>: The proof is done by contradiction. Suppose that there exists a locally valid tree t' with root c such that the distance between t' and $t_\emptyset$ is no greater than th, and t' cannot be obtained with a node operation sequence in $Result$.

Note that if $t' \in L(S')$ and $dist(t_\emptyset, t') \leq th$ then: (i) each subtree $t'_{|i}$ (with $0 \leq i \leq \bar{t'} - 1$) is locally valid, (ii) $dist(t_\emptyset, t'_{|i}) < th$, (iii) $\sum_i dist(t_\emptyset, t'_{|i}) < th$, (iv) the word v formed by t''s root's children is valid *w.r.t.* FSA_c. Note that lines 4–5 in $correctionState$ guarantee that we test all outgoing transitions for every state reached in FSA_c. Thus,

$$correctionTransition(\,t_\emptyset, S, th{-}1, c, M^c_\epsilon,\\ s', t'(0), Result_0)$$

will have to be called. By hypothesis, the sequence nos_0 leading to the subtree $t'_{|0}$ must be contained in the $Result$ of this call. Then,

$$correctionTransition(\,t_\emptyset, S, th{-}MinCost, c, M^c_{t'(0)},\\ s'', t'(1), Result_1)$$

will have to be called with $0 < MinCost \leq cost(nos_0) < th$. By hypothesis, the sequence nos_1 leading to subtree $t'_{|1}$ must again be contained in the $Result_1$ of this call. The same holds for all subtrees of t'. Thus, t' can be obtained from $t_\emptyset$ by the operation sequence

$$nos = \langle\,(insert, \epsilon, c)._{th} AddPrefix(nos_0, 0)._{th}\\ AddPrefix(nos_1, 1)._{th} \cdots ._{th}\\ AddPrefix(nos_{\bar{t'}-1}, \bar{t'} - 1)\rangle$$

This sequence will necessarily be created in $correctionTransition$, line 9, and further added to $Result$ in $correctionState$, line 3.

$\square$

Let us now admit that $th = 0$. This case is considered separately because it leads to no matrix creation.

Lemma 2. Given a tag $c \in \Sigma$, a schema S, and a tree t, let S' be the schema $(\Sigma, c, S.Rules)$. A call to correction(t,S,0,c) always terminates and returns the set *Result s.t.* $t \xrightarrow{Result} L^0_t(S')$.

Proof. Note that $L^0_t(S') = \{t' \mid t' \in L(S')$ and $dist(t, t') = 0\}$. This set contains only t if t is locally valid, and no tree otherwise. Note also that the call to $correction(t, S, 0, c)$ terminates: (i) in line 3 with $Result = \{nos_\emptyset\}$ if t is valid, (ii) in line 6 with $Result = \emptyset$ otherwise. In case (i) $Result$ leads from t to t itself, and in case (ii) to no tree. Thus, the lemma holds. $\square$

Let us now consider any non empty tree t to be corrected with a positive threshold. We will show that each cell of the distance matrix computed by our algorithm transforms partial trees of t into partially valid trees within the threshold th. This can be formally expressed by the following lemma.

Lemma 3. Let $c \in \Sigma$ be a tag, S be a schema, $t \neq t_\emptyset$ be a tree, and $th > 0$ be a threshold. Let $u \in \Sigma^*$ be a word such that $u \in L(FSA_{S.root})$ and $L^c_u(S) \neq \emptyset$. The call to $correction(t, S, th, c)$ computes the matrix M^c_u such that, for each $0 \leq i \leq \bar{t}$ and for each $0 \leq j \leq |u|$, the following proposition holds:

$$t\langle i{-}1\rangle \xrightarrow{M^c_u[i][j]} \{t' \mid t' \in L^c_{u[1..j]}(S), dist(t\langle i{-}1\rangle, t') \leq th\}$$

Proof. Firstly, note that for a non-empty tree t and a positive threshold $0 < th$ a distance matrix is necessarily created (function *correction*, line 10) and filled out.

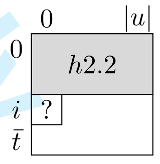


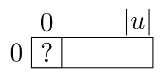
FIGURE 16. Example of matrix representation for the proof

Secondly, let's consider the case of $i = 0$ and $j = 0$. Note that for a non-empty tree t the cell $M^c_u[0][0]$ can only be filled out in the function *correction*, lines 15 and 17. Each of these 2 actions clearly terminates and results in an operation sequence transforming t's root (i.e. $t\langle -1\rangle$) into a root-only tree $\{(\epsilon, c)\}$ with cost no higher than 1. Note also that $L^c_{u[1..0]}(S) = L^c_\epsilon(S) = \{(\epsilon, c)\}$. Thus, the proposition holds for $i = 0$ and $j = 0$.

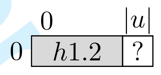
The proof for the remaining cases is done by induction on the depth of the tree t, then on the row index i, and finally on the column index j. We will use the representation of the distance matrix M^c_u as in Figure 16 to say that we want to verify the cell which contains the question mark '?' knowing that the cells in gray are concerned by the hypothesis.

1. Basis ($depth(t) = 1$): If $depth(t) = 1$ then t contains only the root tag, i.e. $t = \{(\epsilon, x)\}$ and $\bar{t} = 0$. Consequently, there exists only one i *s.t.* $0 \leq i \leq \bar{t}$, namely $i = 0$. Thus, we only need to show that for each $0 \leq j \leq |u|$:
$t\langle -1\rangle \xrightarrow{M_u^c[0][j]} \{t' \mid t' \in L^c_{u[1..j]}(S), dist(t\langle -1\rangle, t') \leq th\}$

Further proof is done by induction on column index j.



1.1. Basis ($depth(t) = 1, i = 0, j = 0$): If $j = 0$ then we are considering the same case as above, i.e. $i = 0$ and $j = 0$. We have already shown that the proposition is true for this particular case.



1.2. Induction step ($depth(t) = 1, i = 0, 0 < j$): Suppose that the proposition is true for a $0 \leq j' = j-1$ **(h1.2)**, i.e.
$t\langle -1\rangle \xrightarrow{M_u^c[0][j-1]} \{t' \mid t' \in L^c_{u[1..(j-1)]}, dist(t\langle -1\rangle, t') \leq th\}$.
We will prove that it also holds for j.

Note that with $0 < j$ the cell $M_u^c[0][j]$ can only be filled out in function *correctionTransition*, line 9. The contents of this cell stems from the threshold-bound concatenation of node operation sequences in: (i) the cell $M_u^c[0][j-1]$, (ii) the result of correcting an empty tree with an appropriately diminished threshold, and with target root u_j. This situation is depicted in Fig. 17. By hypothesis $h1.2$, the calculation of (i) terminates and each element in (i) transforms $t\langle -1\rangle$ into a partially valid tree with word $u[1..(j-1)]$ formed by the root's children. By Lemma 1 the calculation of (ii) terminates and each element of (ii) creates a locally valid tree with root u_j. Thus, each concatenation of these elements creates a tree whose:

- root is c,
- root's children form the word $u[1..j]$
- all root's subtrees are locally valid.

In other words this tree is partially valid. The threshold-bound concatenation guarantees that its distance from t is no greater than th. That proves the termination and the soundness of the lemma. The completeness can be proved similarly to Lemma 1. In each $t' \in L^c_{u[1..j]}$ all subtrees are locally valid, have the appropriate distance from t and $u[1..j] \in FSA_c$. Thus, each transition in FSA_c labeled with u_k ($1 \leq k \leq j$) has to be followed, and by hypothesis each subtree $t'_{|k}$ has to be reachable by a sequence stemming from a call to *correctionTransition*. The whole tree t' can be obtained by concatenating the root correction operation with all such sequences for $1 \leq k \leq j$ (prefixed by k).

Thus, the proposition holds for any $0 \leq j$ with $i = 0$.

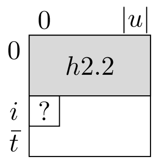
2. Induction step ($depth(t) > 1$): Suppose that the proposition is true for any tree t' with $0 \leq depth(t') < d$ **(h2)**. We will prove that it also holds for any tree t with $depth(t) = d$.

The proof is done by induction on the row index i.

2.1. Basis ($depth(t) > 1, i = 0$): With $i = 0$ we need to show that for each $0 \leq j \leq |u|$:
$t\langle -1\rangle \xrightarrow{M_u^c[0][j]} \{t' \mid t' \in L^c_{u[1..j]}(S), dist(t\langle -1\rangle, t') \leq th\}$.
The proof is the same as in the case of a tree of depth 1 (see above).

2.2. Induction step ($depth(t) > 1, i > 0$): Suppose that the proposition is true for any $0 \leq i' < i$ **(h2.2)**. We will prove that is also holds for i. The proof is done by induction on column index j.

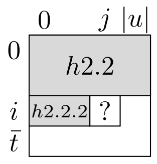
2.2.1. Basis ($depth(t) > 1, i > 0, j = 0$): With $j = 0$ we need to show that:
$t\langle i-1\rangle \xrightarrow{M_u^c[i][0]} \{t' \mid t' \in \{(\epsilon, c)\}, dist(t\langle i-1\rangle, t') \leq th\}$



Recall that cell $M_u^c[0][0]$ contains at most one operation leading to the correct root c. Note also that all other cells in the first column can only be filled out in function *correction*, line 19. The contents of each of these cells stems from the threshold-bound concatenation of: (i) removing a subtree in t, (ii) node operation sequences in the cell above. Thus, the cell $M_u^c[i][0]$ contains at most one operation sequence which represents removing all subtrees in $t\langle i-1\rangle$ (from right to left) and possibly relabeling the root, as depicted in Fig. 18. Obtaining this sequence (if any) clearly terminates and leads to the root-only tree $\{(\epsilon, c)\}$, which proves termination and soundness.

The completeness is straightforward since the only possible element (if any) in $\{t' \mid t' \in \{(\epsilon, c)\}, dist(t\langle i-1\rangle, t') \leq th\}$ is the root-only tree $\{(\epsilon, c)\}$. This tree can be obtained precisely by the unique operation sequence (if any) described above.

2.2.2. Induction step ($depth(t) > 1, i > 0, j > 0$): Suppose that the proposition is true for any $0 \leq j' = j-1$ **(h2.2.2)**.



We will prove that it also holds for j. Note that if $i > 0$ and $j > 0$ the cell $M_u^c[i][j]$ can only be filled out in procedure *correctionTransition*, lines 14–17. Three cases are to be examined.

- The horizontal correction (line 15) yields threshold-bound concatenations of: (i) the cell $M_u^c[i][j-1]$, (ii) the result of correcting an empty tree with an appropriately diminished threshold, and with target root u_j. By hypothesis $h2.2.2$, the calculation of (i) terminates and each element (if any) in (i) transforms $t\langle i-1\rangle$ into a partially valid

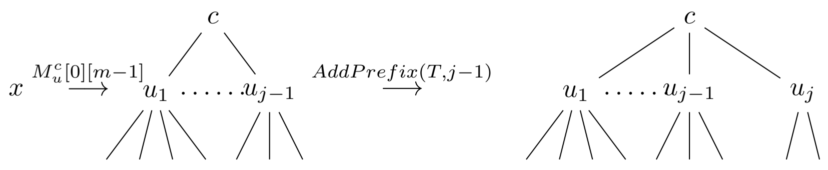
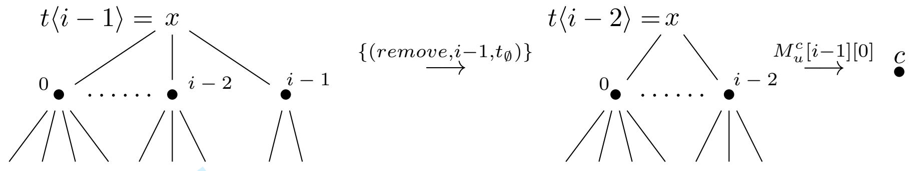


FIGURE 17. Combining operation sequences in case of $i = 0$



$t\langle i-1\rangle = x$
0
$i-2$
$i-1$
$\{(remove, i-1, t_\emptyset)\}$
$t\langle i-2\rangle = x$
0
$i-2$
$M_u^c[i-1][0]$
c

FIGURE 18. Combining operation sequences in the first column of the distance matrix

tree with word $u[1..j-1]$ formed by the root's children. By Lemma 1 the calculation of (ii) terminates and each element (if any) of (ii) creates a locally valid tree with root u_j. Thus, each concatenation of elements in (i) and (ii), provided that it does not exceed the threshold, creates a tree whose:

- root is c,
- the root's children form the word $u[1..j]$
- all root's subtrees are locally valid.

In other words this tree is partially valid. The threshold-bound concatenation guarantees that its distance from $t\langle i-1\rangle$ is no greater than th. That proves the termination and the soundness of this case.

- The diagonal correction (line 16) yields threshold-bound concatenations of: (i) the cell $M_u^c[i-1][j-1]$, (ii) the result of correcting subtree $t_{|_{i-1}}$ with an appropriately diminished threshold th', and with target root u_j. By hypothesis $h2.2$, the calculation of (i) terminates and each element (if any) in (i) transforms $t\langle i-2\rangle$ into a partially valid tree with word $u[1..j-1]$ formed by the root's children. Note that for (ii) two cases are possible. Firstly, th may be equal to 0. In that case, by Lemma 2, the result of (ii) is either empty ($t_{|_{i-1}}$ is not locally valid) or equal to $t_{|_{i-1}}$ (otherwise). Secondly, th may be positive. In that case, each element in the result of (ii) necessarily stems from the cell $M_v^{u_j}[t_{|_{i-1}}^{-}][|v|]$ for a certain $v \in FSA_{u_j}$ (see procedure *correctionState*, line 3). Since the subtree $t_{|_{i-1}}$ necessarily has a smaller depth than t, by hypothesis $h2$, (ii) terminates and
$$t_{|_{i-1}} \xrightarrow{M_v^{u_j}[t_{|_{i-1}}^{-}][|v|]} \{t' \mid t' \in L_v^{u_j}(S), dist(t_{|_{i-1}}, t') \le th\}$$
In other words, each element in (ii) transforms $t_{|_{i-1}}$ into a locally valid tree with root u_j. Thus, each concatenation of elements in (i) and (ii), provided that it does not exceed the threshold, again creates a partially valid tree within the threshold, as depicted in Fig. 19. That proves the termination and the soundness of this case.
- The vertical correction (line 17) yields threshold-bound concatenations of: (i) removing subtree $t_{|_{i-1}}$, (ii) node operation sequences in the cell $M_u^c[i-1][j]$. Clearly, (i) terminates. By hypothesis $h2.2$, the calculation of (ii) terminates and each element in (ii) transforms $t\langle i-2\rangle$ into a partially valid tree with word $u[1..j]$ formed by the root's children. This process, when preceded by deleting subtree $t_{|_i}$ transforms the partial tree $t\langle i-1\rangle$ into a partially valid tree. That proves the termination and the soundness of this case.

We prove the completeness by contradiction. Let's suppose that there exists a tree $t' \in L_{u[1..j]}^c(S)$ such that $dist(t\langle i-1\rangle, t') \le th$, and no operation sequence in $M_u^c[i][j]$ leads from $t\langle i-1\rangle$ to t'. According to the tree distance definition in [17], t' can be obtained from $t\langle i-1\rangle$ by at least one of the three types of corrections (horizontal, diagonal, or vertical correction).

- In the horizontal correction: (i) the partial tree $t\langle i-1\rangle$ is transformed into the partial tree $t'\langle j-2\rangle$, (ii) the subtree $t'_{|_{j-1}}$ is inserted. By hypothesis $h2.2.2$ the cell $M_u^c[i][j-1]$ must contain the sequence allowing to obtain $t'\langle j-2\rangle$ since this partial tree is within the threshold and its root's children form the word $u[1..j-1]$. Note also that, by Lemma 1, the subtree $t'_{|_{j-1}}$ must be reachable by a sequence calculated in *correctionTransition*, line 11. Thus, at least one sequence leading to t' must be obtained.

- In the diagonal correction the partial tree $t\langle i-2\rangle$ is transformed into the partial tree $t'\langle j-2\rangle$ and the subtree $t_{|_{i-1}}$ is transformed into the subtree $t'_{|_{j-1}}$. By hypotheses $h2.2$ and $h2$ at least one corresponding operation sequence must be obtained.

- In the vertical correction the partial tree $\langle i -$

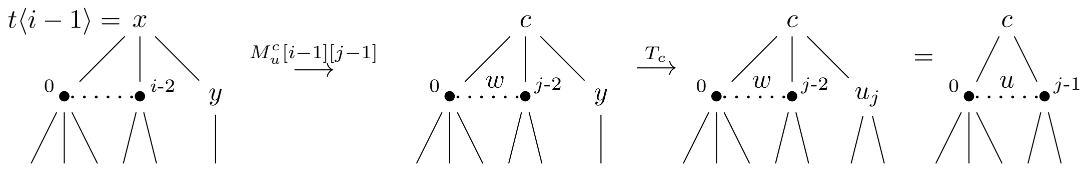


FIGURE 19. Combining operation sequences in case of a diagonal correction with $w = u_1 \dots u_{j-1}$

2⟩ is transformed into t' and the subtree $t_{|i-1}$ is deleted. By hypothesis $h2.2$, and by line 17 in *correctionTransition*, at least one corresponding operation sequence must be obtained.

□

THEOREM 1. Given a tree t, a schema S and a threshold $0 \le th$, a call to $correction(t, S, th, S.root)$ terminates and returns a set *Result* such that the following proposition holds:
$t \xrightarrow{Result} L_t^{th}(S)$.

Proof. If $th = 0$ then the proposition holds by Lemma 2. Suppose now that $0 < th$.

(*a*) <u>*Termination*</u>: The proof is done by induction on the depth of the tree t. If $depth(t) = 0$ the tree is empty and, by Lemma 2, the call to $correction(t_\emptyset, S, th, S.root)$ terminates. Suppose now that the call terminates for each tree t' *s.t.* $0 \le depth(t') < d$. We will prove that it also terminates for a tree t with depth d. With $0 < th$ the set *Result* can receive new operation sequences only in the procedure *correctionState*, line 3. These corrections stem from the matrix $M_u^{S.root}$ for some $u \in FSA_{S.root}$. By Lemma 3, the calculation of each matrix cell terminates. Note that the recursion is induced in *correctionTransition* by lines 8, 13, 16 and 22. The recursion in lines 8 and 13 terminates by Lemma 1. The recursion in line 16 terminates by the induction hypothesis since the subtree $t_{|i-1}$ has a smaller depth than t. Moreover the cells filled out in lines 9, 15 and 17 necessarily have a higher cost than the cells they have been deduced from ($M_v^c[0][m-1]$, $M_v^c[i][m-1]$ and $M_v^c[i-1][m]$, respectively) because at least one node insertion or deletion is concatenated. Only the concatenation in line 16 can lead to sequences of the same cost as in the preceding cell $M_v^c[i-1][m-1]$. That can however happen only for locally valid subtrees, whose number is bounded by $\bar{t}$. Thus after at most $\bar{t} + th$ recursive calls between procedures *correctionState* (line 5) and *correctionTransition* (line 22) all solutions in the current column must have a cost exceeding th and the recursion stops.

(*b*) <u>*Soundness*</u>: With $0 < th$ the set *Result* can receive new operation sequences only in the procedure *correctionState*, line 3. These corrections stem from the bottom right-hand cell of the matrix, i.e. $M_u^{S.root}[\bar{t}][|u|]$, for some $u \in FSA_{S.root}$. By Lemma 3 we have:
$t\langle\bar{t}-1\rangle \xrightarrow{M_u^{S.root}[\bar{t}][|u|]} \{t' \mid t' \in L_u^{S.root}(S), dist(t\langle\bar{t}-1\rangle, t') \le th\}$.
Since $t\langle\bar{t}-1\rangle = t$ we get:
$t \xrightarrow{M_u^{S.root}[\bar{t}][|u|]} \{t' \mid t' \in L_u^{S.root}(S), dist(t, t') \le th\}$
Note that $L_u^{S.root}(S) \subseteq L(S)$, thus each element in *Result* leads to a tree that belongs to $L_t^{th}(S)$, which proves the soundness.

(*c*) <u>*Completeness*</u>: The proof is done by contradiction. Suppose that there exists a tree $t' \in L_t^{th}(S)$ such that no operation in *Result* leads from t to t'. Let v be the word formed by the children of t's root, i.e. $v = t'(0)\dots t'(\bar{t'}-1)$. Note that $t'(\epsilon) = S.root$ since t' is valid. The function *correction* necessarily creates a matrix $M_\epsilon^{S.root}$ in line 10, fills out its first column and the cell $M_\epsilon^{S.root}[0][0]$ contains the node operation leading from $t(\epsilon)$ to $S.root$. Further on, the procedure *correctionState* necessarily tests the transition in $FSA_{S.root}$ labeled with $t'(0)$. By Lemma 3 the second column in $M_{t'(0)}^{S.root}$ necessarily contains sequences leading from partial trees of t to the partial tree $t'\langle 0\rangle$. By definition of the tree distance, at least one of these sequences has a cost no higher than th. Thus, necessarily the procedure *correctionState* is again called by procedure *correctionTransition*, line 22 and the transition labeled with $t'(1)$ is examined. These calls continue until $t'(\bar{t'}-1)$ and each new column contains at least one sequence within the threshold. After the transition labeled with $t'(\bar{t'}-1)$ is examined the contents of the cell $M_v^{S.root}[\bar{t}][\bar{t'}]$ is added to *Result*. By the proof of soundness, this cell leads from t to t'.

□

4.3. Complexity

In the correction process there are two sources of recursion: the structure description S on the one hand and, on the other hand, the input tree t. Both are traversed in parallel, except for the correction of an empty tree.

Indeed, the recursion within the structure description is the only one used by the correction of an empty tree to a target root label a. It corresponds to the insertion of a node with label a, this node being the root of a tree.

For inserting a tree with root label a, we consider the rule for a in S and we try to build correct children words, i.e. we try to insert children nodes, those nodes being themselves potential roots of trees. Thus, for each new child node we consider again the corresponding rule in S and try to build correct children words, etc.

The process of correcting an empty tree to a target root label a is bounded by th. Indeed, as the maximum size of correct trees is th, the process must stop as soon as th nodes have been inserted (into the empty tree). Thus, the maximum number of transitions followed in S (in all relevant automata) is always bounded by $(f_S)^{th}$, where f_S is the maximum fan-out of all states in all finite state automata in the structure description S.

Notice that the process of correcting an empty tree to a target root label a may stop much sooner than this general limit. For instance if FSA_a contains only one state, the final one, the process stops immediately. It is clear that the shape of schemas, i.e. the form of automata in S, is an important parameter of complexity: obviously, the less there are choices, the lower is f_S. Repetitions in the schema (i.e. cycles in automata in S) imply potentially infinite length of correct children words. Recursion in the schema also induces potentially depth-infinite valid trees. These two latter cases are limited by the threshold th.

The second source of recursion is the input tree t. Each of its nodes may need to be corrected, thus a complete recursive traversal is necessary. This is done with recursive calls to the correction process when computing correction matrices.

Considering correction matrices, we can fix a general bound for: (i) the number of their rows, and (ii) the maximum number of computed columns, for all the matrices taken together.

For each computed matrix, its *number of rows* is bounded by the maximum number of children of all nodes in the input tree, denoted f_t, plus one (for the root).

The *number of columns* computed for correcting t depends on the dynamic exploration of the FSAs in S.

Considering the exploration graph illustrated in Figure 4, for one input word w the length of all correct words is less than or equal to $|w|$ plus th. Moreover, for each node in the exploration graph there is at most f following nodes to explore, where f denotes the maximum fan-out of states in the FSA. Thus, the *number of columns* computed in Oflazer's algorithm is bounded at worst by $f^{|w|+th}$.

Generalizing this fact to the tree context, the limit for the length of correct words, $|w| + th$, becomes $|t| + th$, the maximum size of correct trees. Thus, *the maximum number of columns computed in matrices taken all together* is bounded at worst by $(f_S)^{|t|+th}$.

We have the following bound for the number of computed cells: $(f_t + 1) \times (f_S)^{|t|+th}$. Now each cell computation consists in inserting, deleting or renaming one node. The recursion on the subtree rooted at this node is taken into account in the global number of columns times the maximum number of rows. Nevertheless, as we compute *all* solutions within the threshold th, for each cell we concatenate sets of node operation sequences. Let's try to bound number of such concatenations.

The length of each node operation sequence included in the matrix is always bounded by th. Recall that a node operation on tree t is a triple (op, p, a) we have:

- For $op = delete$: p belongs to t.
- For $op = relabel$: a belongs to Σ; p belongs to t.
- For $op = add$: a belongs to Σ; p belongs to the set of positions $Pos(t) \setminus \{\epsilon\} \cup InsFr(t)$, whose size is equal to 1 for an empty tree, and to $2 \times |t| - 1$ for a non empty tree[7].

Thus, the possible choice for p is always the largest in case of a sequence containing node additions only. Note also that when a node addition is performed on t a new position is created in t. Thus, a new node addition (if any) can appear in any of the resulting $2 \times (|t| + 1) - 1$ positions. Yet another node addition could concern one of the $2 \times (|t| + 2) - 1$ position, etc. In total, the number of possible sequences of positions where node operations intervene is bounded by $(2 \times (|t| + th))^{th}$. For any of these positions we can choose among one of at most three operation types, and one of at most $|\Sigma|$ target labels. Therefore, the size of sets of all possible node operation sequences is bounded by $(3 \times |\Sigma| \times 2 \times (|t| + th))^{th}$.

We can conclude that the time complexity of our tree correction algorithm is in
$O((f_t + 1) \times (f_S)^{|t|+th} \times 6 \times |\Sigma| \times (|t| + th))^{th}$.

In practice two facts decrease the previous bounds, which are difficult to formalize in a complexity analysis. Firstly, the dynamic FSA exploration is fast limited by the threshold th: we stop adding a new column as soon as all cells in the current columns are empty (a cell becomes empty if its corresponding cost is greater than th). Secondly, as in Oflazer's proposal for strings, the computations of partial trees are factorized, i.e. the initial columns are computed only once for all words having the same prefix. These may be the reasons why the experimental results described in the following section hardly ever show an exponential cost in time, whether in function of the document size or of the threshold.

Moreover, some optimizations may be introduced in the implementation, in order to decrease the computation time. One of them consists in saving the intermediate correction results in an auxiliary structure in such a way that the correction of a subtree of t for the target root label a and with the threshold th is performed at most once. It is detailed in Appendix A. For documents that contain sequences of large subtrees,

[7]Note that the insertion frontier contains exactly one new child position for each node existing in the initial tree.

with identical structures, this intermediate storage largely increases the correction performances.

4.4. Adaptations and Extensions

The presented approach to correct an XML document t *w.r.t.* a schema S is unique in its *completeness* in the sense that, given a non negative threshold th, the algorithm finds every tree t' valid with respect to S such that the edit distance between t and t' is no higher than th. In this way it offers the guarantee of getting all solutions verifying a precise criterion (here: within a given threshold). This allows us to be sure that the best solution (satisfying the criterion) is actually in the resulting set. Notice that features defining what exactly is *the best solution* highly depend on application contexts and sometimes may be rather informal (thus difficult to select automatically). Nevertheless, providing a potentially large set of solutions, in particular to an end-user, is generally counterproductive. For this reason, each application based on our approach would have to perform post-processing tasks in order to take advantage of the computed solutions in a way adapted to the particular context. Context-dependent pre-processing tasks might also be useful. The availability of the source code under an open license allows these kinds of adaptations.

As an example, when a schema update invalidates several documents (cf Section 6.4), most of them may be concerned by the same selected corrections. In this case, an accurate application might use our solution in order to explore the alternative corrections for a single document only. Then the application may ask a user to select the preferred correction, and apply the chosen edit script to all other documents concerned.

Since the proposed algorithm finds all corrections within a threshold, the unavoidable question is how to choose the particular threshold value. On the one hand, if this value is unnecessarily high, the processing time might be unacceptable. On the other hand, if it is not high enough, the algorithm might fail to find the proper correction (or any correction). Here again, the answer highly depends on the context of use. For a given application, a testing scenario such as the one in Section 5 might be designed in order to estimate the optimum threshold value for a given document size. This estimation might start with determining the edit distance between t and $L(S)$, which is efficiently computable with one of the algorithms specifically designed for this problem (cf. Section 6.1).

Our algorithm can also be used for estimating the optimum th, with the following scenario: if the document is not valid then the distance is at least 1, thus search corrections with $th = 1$; if no correction of cost 1 is returned then search corrections with $th = 2$, etc., until the first non-empty solution set is returned[8].

[8] Our tool uses this scenario when the user asks for the minimal corrections only.

One good reason for applying this scenario is, again, that we are sure of what the algorithm computes. One disadvantage is the theoretical complexity of such a setting, that remains exponential thus higher than in other algorithms capable of finding the document-to-schema distance. However, as shown in the sixth testing scenario in Section 5, the experimental behavior of such a solution is polynomial rather than exponential.

Advantages and drawbacks of computing all solutions satisfying a well defined criterion are a point of discussion in many other fields, as for instance the field of strings. Indeed, not far from our concerns, the recent exciting survey in [20] shows how the approximate string matching problem has received attention from many different scientific communities, which frequently have built comparable algorithms, sometimes in parallel, for their specific practical application contexts. Considering the place now taken by XML in all information systems, it can be assumed that all the situations in which tree-to-language correction will be useful[9] are not known yet. Thus, a deliberately application-independent approach, as the one proposed in this article, although it may appear too complete, is a valuable contribution.

We now present extensions that are not yet implemented in the tool associated with this article. The first one concerns the schema languages that our algorithm can deal with. We have initially focused on DTDs, which correspond to Local Tree Grammars in the taxonomy of [22], but we show that dealing with Single Type Tree Grammars, i.e. with schemas expressed using XML Schema (XSD), is a straightforward extension.

In a schema defining a Local Tree Language (such as a DTD), each element name is associated with a unique content definition (or type). In a schema defining a Single Type Tree Language, it is possible to have more than one type for the same element name, provided that the so-called *competing* types are never used simultaneously in the same definition type. In other words, in an XSD schema one can associate different type definitions with the element name, but they are distinguished by their context (they can be seen as local definitions). Thus, the only difference with respect to the case of DTDs is that one has to resolve competition of types by using type definitions of *parent nodes*.

Our algorithm can easily be adapted to deal with that class of schemas since the recursion goes top-down. For the root node, it is guaranteed that competition of types for a given label never occurs (in an XSD schema, only globally defined elements can be root nodes and it is forbidden to define two global elements with the same name but different types). When correcting a nonroot node, we already know its parent-node type definition,

[9] Extended with pre-processing and post-processing tasks depending on the context.

thus we can uniquely determine the schema rule for this node in this context. As there are several possible root labels, the initial correction thread should be triggered for each one but this adds only a multiplicative factor to the overall complexity.

More precisely, the only changes that are necessary for dealing with schemas defining Single Type Tree Languages are the following ones:

- In the schema structure S, we associate each rule not only with a label a, but with a triple (p, u, a), where p and u are paths of labels, $p.u$ representing the context of the definition of a. If there is a finite number n of non-recursive definitions, we will have n triples (p_i, ϵ, a), $1 \leq i \leq n$, where p_i denotes a path of labels from a root node to the label a. Otherwise, for each infinite set R of paths sharing a repeated pattern r due to recursive definitions, we will have one triple (p, r, a), with p being the common prefix of paths in R and r being the repeated pattern. In this way, the set *Rules* in S becomes a set of pairs $((p, u, a), FSA_{(p.u,a)})$.
- The element *root* in S becomes a set *Root* $\subseteq \Sigma$.
- In the algorithm, we must add p' (string of labels) as a parameter of the function *correction* and also of each procedure. Similarly to $p.u$ above, p' contains the path between the root of the XML document and the current node, which is the root of the subtree t to be corrected. Using p', it is then possible to choose in S the FSA for correcting t.
- The function *correction* must be iteratively called for each element of the set of possible root labels. For each one, it is initially called with $p' = \epsilon$.
- In the procedure $correctionTransition$, every call to the function *correction* is done with $p'.t(\epsilon)$. Thus, the label of the root of t is added to the path representing the context, for each subtree correction.

Other extensions concern time and space optimizations of our algorithm, inspired by related works. Firstly, recall that each cell in our distance matrix M^c_u (cf Section 4.1) is calculated on the basis of its three upper-left-hand neighbors. Thus, following the idea reported by [20] and implemented by [4], only two consecutive columns have to be stored at a time. Note however that backtracking might impose recalculating some previously deleted columns. Secondly, as reported again by [20], many optimizations have been proposed to speed up the string-to-language correction implementations. Since our algorithm extends this problem, some of these optimizations might apply, as e.g. bit-parallelism (simultaneous updates of numbers packed into a single computer word) or FB-tries (evoked at the end of Section 5).

Another interesting perspective would be to enlarge the diversity of elementary edit operations and their costs. For instance, we might consider (similarly to [3], [23] and [2]) the possibility of inserting or deleting internal tree nodes, not necessarily leaves. We think that this problem is related to the *extended edit distance* on strings [20], in which substitutions of arbitrary strings rather than single-character edit operations are allowed. In the context of trees, inserting/deleting an internal node would correspond to replacing a sequence of sibling roots by a new node, or vice versa. Also, exchanging subtrees or moving them within the whole tree in a single operation might yield a good modeling of document proximity in various application domains, as discussed in the tree-to-tree correction literature, e.g. [24] and [25]. Finally, using operation costs dependent on the positions where the operations apply might be worthwhile, as shown in [20].

5. EXPERIMENTAL RESULTS

Several experiments were conducted in order to examine the performances of our algorithm on real-life data in function of different parameters: (i) the document size, (ii) the threshold value, (iii) the number of errors, (iv) the position of an error, (v) the nature of the DTD. In this section we describe the settings of six testing scenarios and we provide their results.

We have used a large XML file, henceforth called the *test file*, containing linguistic annotations of named entities performed within the Polish National Corpus project [26]. The test file is compliant with the DTD presented in Fig. 20. This DTD seems rich enough to cover different phenomena that might influence the performances of the correction algorithm. Namely, it defines elements concerned by a varying degree of flexibility (which potentially yields a varying number of corrections).

The metadata, i.e. the two first subelements, ⟨`head`⟩ and ⟨`meta`⟩, as well as their three child nodes ⟨`schema`⟩, ⟨`id`⟩ and ⟨`subId`⟩, correspond to the part of the DTD in which optionality, alternative or unbounded repetitions of elements are not admitted (i.e. no `?`, `|`, `+` and `*` are used in the respective regular expressions). Thus, if an error is introduced in this part of the test file, there can hardly be any ambiguity about how this error should be corrected. Henceforth this part of the test file will be called the *nonambiguous part*.

The rest of the test file (henceforth the *ambiguous part*) is composed of 2638 ⟨`sent`⟩ences divided into ⟨`seg`⟩ments (i.e. roughly words) and named entities (⟨`ne`⟩). Each ⟨`seg`⟩ment contains its ⟨`orth`⟩ographic form (i.e. the inflected form appearing in the text, e.g. *domu*) and its ⟨`base`⟩ form (i.e. the lemma, e.g. *dom*). The description of named entities is more complex and allows for optionality, alternative, unbounded repetition of elements and recursion. In particular, the ⟨`children`⟩ of an ⟨`ne`⟩ are ⟨`seg`⟩ments and/or other named entities (e.g. *[ulica [[Kazimierza] [Pułaskiego]]]* '[[[Kazimierz] [Pułaski]] Street]'). That allows for theoretically unbound recursive embedding of named entities. However in practice no more than a

```
<!ELEMENT NKJP_names (head,meta,sent*)>
<!ELEMENT head (schema)>
<!ELEMENT schema EMPTY>
<!ELEMENT meta (id,subId)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT subId (#PCDATA)>
<!ELEMENT sent (seg|ne)+>
<!ELEMENT seg (orth,base)>
<!ELEMENT orth (#PCDATA)>
<!ELEMENT base (#PCDATA)>
<!ELEMENT ne (((orth, base)|(when, orth)),
              (derivType, (derivedFrom)?)?,
               cert, certComment?, children)>
<!ELEMENT when (#PCDATA)>
<!ELEMENT derivType (#PCDATA)>
<!ELEMENT derivedFrom (#PCDATA)>
<!ELEMENT cert (#PCDATA)>
<!ELEMENT certComment (#PCDATA)>
<!ELEMENT children (seg|ne)+>
```

FIGURE 20. The DTD of the XML file used for experiments

few embedding levels appear (3 levels in our file). As a result, our test XML file has a rather flat structure. There are many sentences and each sentence contains many segments and/or named entities (i.e. the root node has thousands of children and grand children), while segments and named entities are represented by relatively shallow structures (i.e. the depth of the document tree does not exceed 10).

The test file has been transformed in different ways so as to design different testing scenarios. Four input parameters have been taken into account:

- the size of the document to be corrected,
- the distance threshold,
- the number of errors introduced,
- the position at which an error has been introduced (the ambiguous vs. the non ambiguous part, and the beginning vs. the end of the document to be corrected).

Two types of results have been examined:

- the CPU times consumed during the correction process,
- the number of corrections obtained.

The algorithm was implemented in Java 1.6 and the tests were run on an Intel Core i3-2310M 2.10GHx4 machine under Ubuntu Oneiric Linux 11.10, with a 8 GB RAM and a 500 GB hard disk.

The *first* scenario was designed so as to test the correction time and the number of correction candidates obtained in function of the document size. In this experiment:

- The test file was repeatedly reduced into files containing: the metadata, the first and the last sentence, some sentences following the first one. In this way we obtain a series of valid files $f_1^1, \ldots, f_n^1$ such that files f_i^1 and f_{i+1}^1 differ by about 10–20 nodes.
- One error was introduced in the first sentence of every file f_i^1 (a ⟨`base`⟩ element was deleted under a ⟨`ne`⟩ element). Thus, the error appeared at the beginning of each file in its ambiguous part.
- The threshold was $th = 2$.

Fig. 21 shows the results of this scenario. Note that the CPU time has a rather polynomial behavior, despite the theoretical exponential time complexity described in Section 4. We think that this difference might result from a rather rough complexity estimation due to the greatly recursive nature of our algorithm (path prefix factorizations offered by finite-state automata are hard to express in worst-case estimations). Note that the optimization described in Appendix A (storing results of previously performed corrections in an auxiliary structure) has some minor influence in the processing time: this is due to the fact that correcting subtrees with less that 150-200 nodes is not more time consuming than managing the auxiliary structure. In our experimentations, similar subtrees do not have more than 50 nodes. Note also that the CPU time shows some irregularities which closely correlate with the number of corrections found. For instance, for files of sizes 300 through 370 the CPU time grows linearly as long as the number of results is equal to 26. However the CPU time grows rapidly when the number of correction candidates comes up to 30 and 33.

The *second* scenario was meant to show how the value of the distance threshold th influences the correction time. In this experiment:

- The test file was repeatedly reduced into valid files $f_1^2, \ldots, f_n^2$, as in the previous scenario.
- One error was introduced at the beginning of every file f_i^2 in its non ambiguous part (the ⟨`schema`⟩ element was deleted under the ⟨`head`⟩ element).
- The threshold was fixed to $th = 1$, $th = 2$ and $th = 3$ for the three stages of the experiment, respectively.

As shown in Fig. 22 the correction time dramatically grows for bigger documents and higher thresholds. For instance, for a document of 193 nodes the time needed with $th = 3$ is about 20 and 114 times larger than with $th = 2$ and $th = 1$, respectively. This is clearly due to the fact that the threshold value is one of the main factors limiting the search space in the finite state automata explored during the correction process (exploration paths are cut off as soon as the aggregated edit distance of the previously obtained partial solutions exceeds th).

In the *third* scenario we examined the influence of the number of errors introduced in the corrected file on the CPU time consumed during the correction process. In this experiment:

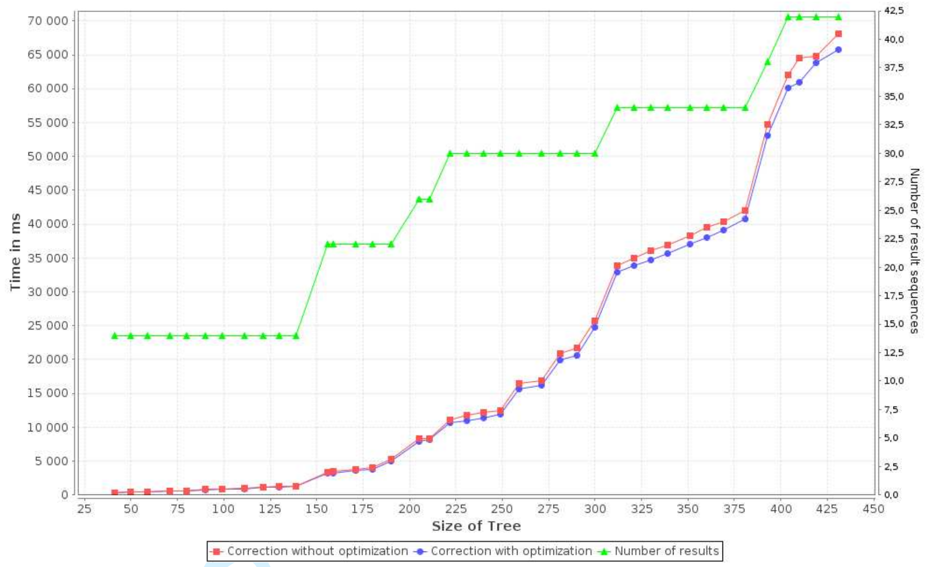


FIGURE 21. CPU time consumed during correction process and number of candidates found in function of the document size (number of nodes) with $th = 2$

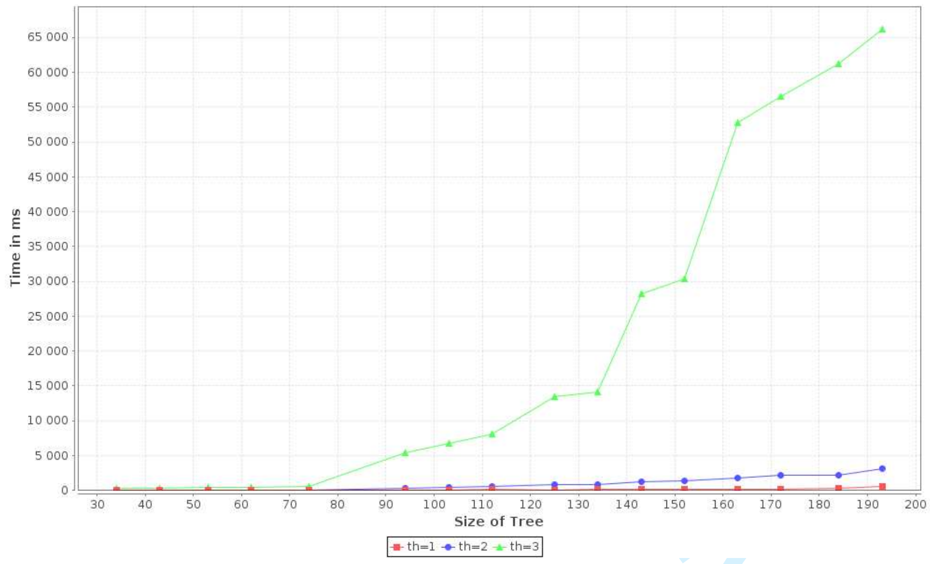


FIGURE 22. CPU time consumed during correction process with different values of the distance threshold, in function of the document size (number of nodes)

- The test file was reduced into a valid file f^3 containing one sentence only and 44 nodes.
- Errors were repeatedly introduced to the file f^3, starting from the leftmost and ending with the rightmost nodes. As a result, twenty one files $f^3_0, \ldots, f^3_{20}$ were produced so that the file f^3_{i+1} had one error more than f^3_i.
- The threshold was fixed to $th = 6$.

As shown in Fig. 23 the CPU time consumption during the correction process is high for files with few errors, and low for files with many errors. This is probably due to the fact that the correction process can continue only if the subtrees placed to the left of the current node could be corrected. Since each partial correction increases the aggregated edit distance, the search space reduces quickly if many errors appear (especially close to the beginning of the file). If more than 6 errors appear there can be no possible correction so the correction time reduces dramatically. Note also that Fig. 23 shows a high correction time for a document with 0 errors, i.e. a valid document. Clearly, if we are interested in correcting a file only when it is incorrect, this result is non relevant. In this case the correction should be preceded by a validation. Our algorithm answers however a more general problem: the one of finding all valid files whose distance from the input file (regardless of its validity) is no bigger than a threshold.

The hypothesis for the *fourth* scenario was that the

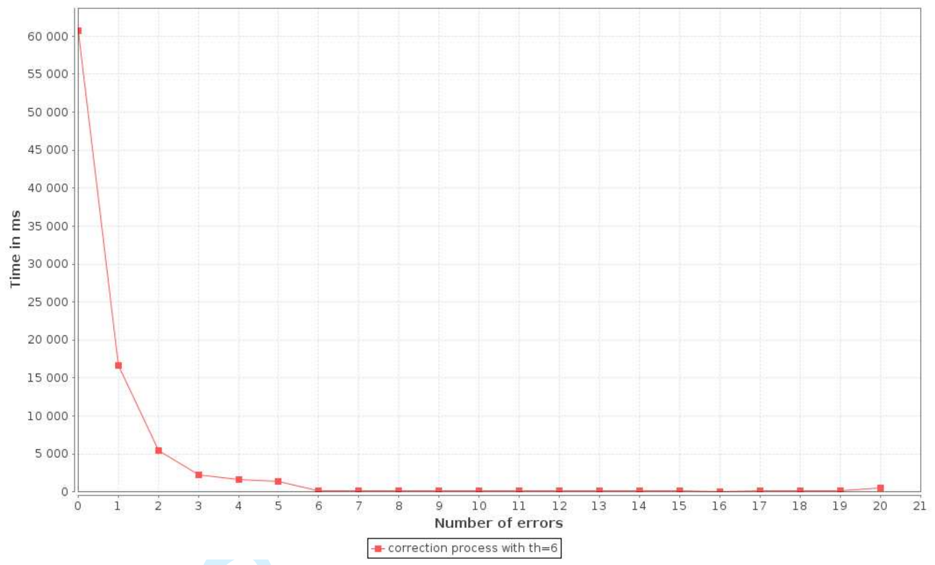


FIGURE 23. CPU time consumed during correction process with $th = 6$ in function of the number of errors in the corrected file

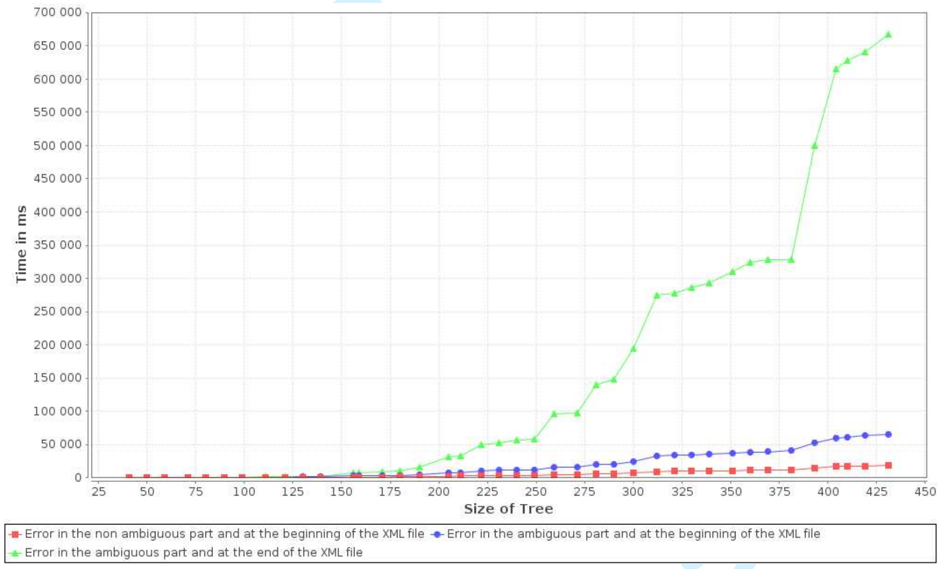


FIGURE 24. CPU time consumed during correction process with an error placed at different positions of the test file, and with $th = 2$, in function of the document size (number of nodes)

position of the error in the corrected file, as well as the nature of the DTD, has an influence on the correction time, and on the number of correction candidates produced. In this scenario:

- The test file was reduced into a valid file f^4 containing 530 nodes.
- The test file was then repeatedly reduced into valid files $f^4_1, \ldots, f^4_n$, as in the first scenario.
- An invalid file $f^4_{(i,1)}$ was created by introducing an error in the metadata of file f^4_i, for each $1 \leq i \leq n$ (the ⟨`schema`⟩ element was deleted under the ⟨`head`⟩ element). Thus, this error appeared at the beginning of the document and was non ambiguous, i.e. there was only one way to correct it.
- Two other invalid files $f^4_{(i,2)}$ and $f^4_{(i,3)}$ were created by introducing an error once in the first sentence, and once in the last sentence of the file f^4_i (the ⟨`base`⟩ element was deleted under the ⟨`ne`⟩ element, i.e. within the file's ambiguous part). These errors were ambiguous, i.e. could be corrected is several ways.
- The threshold was fixed to $th = 2$.

As shown in Fig. 24 the CPU time needed for correction stays relatively low when an error appears at the beginning of the file, while it dramatically grows if the error appears at the end of the file. This is most probably due to the fact that left-hand subtrees

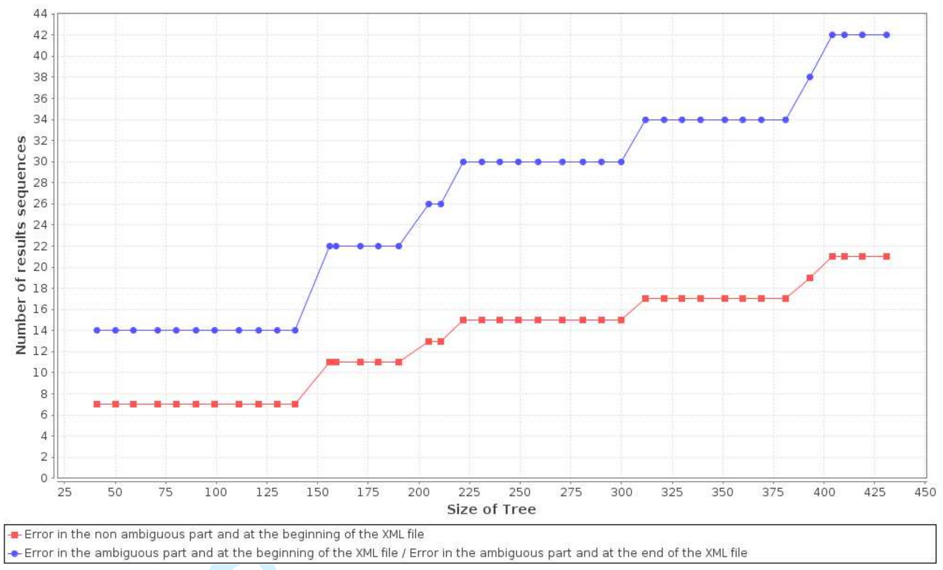


FIGURE 25. Number of candidates found with an error placed at different positions of the test file, and with $th = 2$, in function of the document size (number of nodes)

must be corrected before the current node is examined. Thus, if errors appear close to the beginning of the file the aggregated edit distance increases quickly and the search space gets reduced early. Conversely, if all left-hand subtrees are valid, the aggregated edit distance is equal to 0 and the search space remains limited by the initial threshold only.

Note also (Fig. 25) that the correction for an error appearing in the non-ambiguous part is several times faster than in the ambiguous part. Here again, this is closely correlated with the number of corrections found, which is twice as low in the first case than in the second one due to alternatives allowed by the DTD for the ⟨`ne`⟩ element.

The *fifth* scenario allowed us to examine the influence of the distance threshold on the correction time and the number of candidates found. In this scenario:

- The document to be corrected was empty.
- The threshold increased from 1 to 15.

As shown in Fig. 26 both the correction time and the number of candidates found are of polynomial nature with respect to the threshold value.

The final, *sixth*, scenario was to show how our algorithm behaves when used for determining the set of minimal-cost corrections (for instance in order to help the user choose the accurate threshold needed for her application, as explained in Section 4.4). In this scenario:

- The same set of 21 files (with an increasing number of errors) was used as in the third scenario.
- The algorithm was run repeatedly with $th = 1$, then with $th = 2$, etc., until the first non empty solution set was returned.

As shown in Fig. 27, the experimental behavior of such a scenario is again polynomial rather than exponential.

In conclusion, despite its theoretical exponential time complexity, our algorithm shows a behavior which is rather polynomial in function of the threshold value (Fig. 26), and of the document size (Fig. 21, 22 and 24). Surprisingly enough, the higher the distance of the corrected document from the schema, the shorter the correction time (Fig. 23). This is probably due to the fact that errors appearing close to the beginning of the document rapidly reduce the correction time (Fig. 24), which results from the left-to-right processing of each siblings' level. A possible optimization might be achievable from processing each siblings' word both from left to right and from right to left, as in forward-and-backward tries (FB-tries by Mihov and Schulz) described in [20]. Understandingly, if errors appear in the ambiguous part of the file (i.e. the part concerned by optionality, alternative, and unbounded repetitions of elements in the schema) their correction is more time consuming than in the non-ambiguous part (Fig. 24). Finally, the correction time is closely correlated with the number of correction candidates found (Fig. 21 and Fig. 26), which on its turn directly results from the above-mentioned factors: the size of the input document, the threshold value, the position of an error and the nature of the schema. This correlation between the correction time and the number of candidates might explain, at least partly, why the tree-to-language correction problem, as defined in our approach, is more difficult to solve than in some other works discussed in Section 6. Namely, this problem is frequently reduced in the literature to finding the tree-to-language distance only, without proposing a particular correction sequence, or to proposing a fixed number of minimal sequences only (see Table 1). If completeness of the correction set is

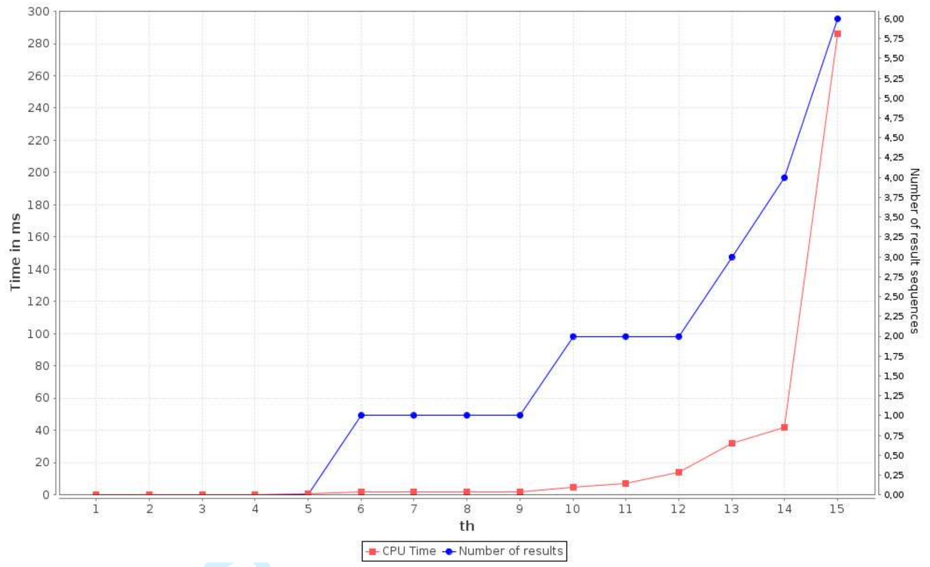


FIGURE 26. CPU time consumed and number of candidates found during the correction of an empty file, in function of the distance threshold

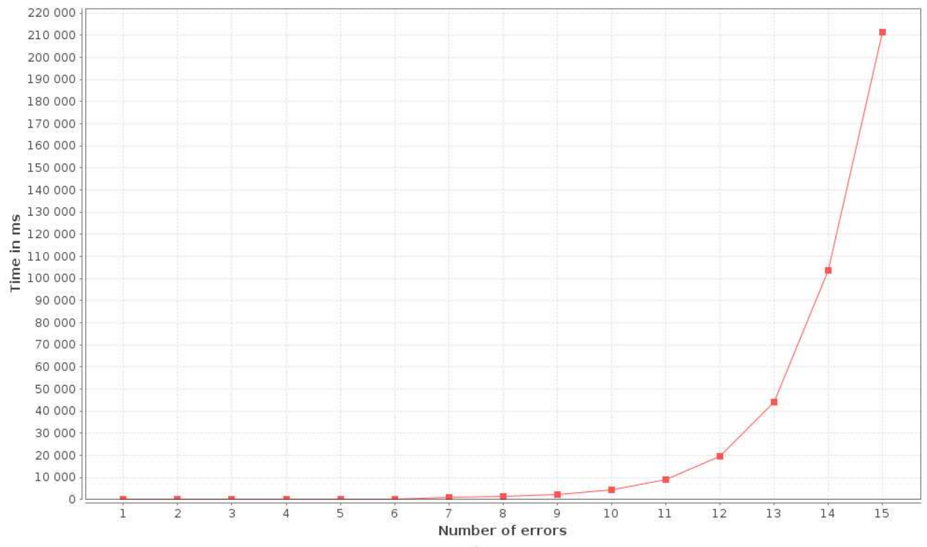


FIGURE 27. CPU time consumed when searching for the tree-to-schema distance, in function of the number of errors in the corrected file

required, the correction time grows accordingly.

6. RELATED WORKS

We resume in this section several definitions of the tree-to-schema correction problem, as well as various corresponding algorithm designs, which exist in the literature. We also provide their contrastive analysis allowing us to situate our own proposal within the state of the art. We see three important degrees in works related to our XML correction algorithm: *(i) measuring a distance between an XML document and a schema*, which is often related to *(ii) finding one or all minimal-cost corrections*, and finally the most complex task of *(iii) computing a set of edit operation sequences for correcting an XML document w.r.t. a schema.* Moreover, *revalidation and correction* intervening *after updates performed on a document or on a schema* may be seen as a particular instance of the tree-to-schema correction problem. We address all of these directions in the following subsections.

6.1. Measuring the Distance Between an XML Document and a Schema

A comprehensive review study dedicated to the problem of XML document/grammar comparison, hence the problem of computing XML document/grammar distance, or similarities, is presented in [1]. The authors present also a complete overview of applications for such computations, which are also potential applications of our work.

In [6, 7] the notion of *structural similarity* between an XML document t and a DTD D is defined and an algorithm to compute this measure is presented. This structural similarity consists in the maximal value of a function that evaluates the set of mappings from t to D. The evaluation relies on computing three parameters, depending on several requirement settings. These parameters are (i) c, the amount of *common* parts, (ii) p, the amount of *extra* parts (i.e. not in the DTD) and (iii) m, the amount of *missing* parts. The algorithm designed to compute this similarity is based on a function $Match$ that computes best mappings, whose complexity is in $O((|t| + |D|) \times a^2)$ where $|t|$ is the document size, $|D|$ is the DTD size and a is the maximal arity of t. Experimental results show good performances. The main advantage of such a measure is its flexibility when a different relative relevance degree can be assigned to each requirement taken into account (such as levels at which common and different features are detected or tag equality versus tag similarity, etc.) that could be set by users depending on the application domain. This is not the case for the proposal in [8], which also measures the structural similarity between an XML document and a DTD, both being modeled as ordered labeled trees. To this aim they adapt a tree-to-tree edit distance measure to the specificity of the special trees they design to represent DTDs, giving a polynomial algorithm for finding the minimum cost sequence of edit operations between the XML tree and the DTD tree, that can be visualized as an explanation of the similarity measure.

At first glance, [5] deals with a problem that is more similar to ours: finding a minimal-cost sequence of elementary edit operations (relabeling a node, and deleting or inserting leaves) allowing the transformation of an XML document into a tree which is valid with respect to a schema. Their considerations are made both on DTDs and XML schemas. Contrary to the initial aim, the algorithm described deals only with the edit distance between a tree and a schema, and not with finding the edit sequence to correct the tree. It is claimed that the algorithm is based on dynamic programming (as ours) and its time complexity is of $O(n \times p \times \log p)$, with n and p being the size of the tree and of the schema respectively. However, the pseudocode does not allow for verification. An interesting point of this paper is the description of an application framework: XML document classification. Convincing arguments are given in favor of using tree-to-schema distance rather than tree-to-tree distance for this application. What is used here however is only the edit distance alone and not the edit sequence, thus, it is hard to understand if the correction procedure has really been studied and implemented.

In [4] the problem of the edit distance between an XML document d and a DTD D is defined, as in our approach, as the minimum cost of all edit sequences transforming d into a D-valid document d'. The elementary edit operations considered are deletions and insertions of a leaf. Complex *macro-operations* defined over these edit operations are (as in our previous work [10]): (i) deleting a subtree, (ii) inserting a minimum-size valid subtree with a given root label, (iii) recursively repairing a subtree. The algorithm is based on a *restoration graph* resembling a matrix with N rows and $M + 1$ columns, where N is the number of states in the finite-state automaton corresponding to the root's label (FSA_r), and M is the number of the root's children. A vertex q_i^j in line i and column j represents both the current state in FSA_r ($0 \leq i < N$) and the current position in the word of the root's children ($0 \leq j \leq M$). Each edge of the graph corresponds to a macro-operation applied on root's children and has the weight equal to the cost of this macro-operation. In particular, establishing the cost of a subtree repair requires a recursive construction of a restoration graph for the current subtree. The problem of finding the document-to-DTD edit distance is thus reduced to finding the shortest path in the restoration graph from vertex q_0^0 to any vertex representing an accepting state in the last column. The paper indicates the time complexity of $O(|D|^2 \times |T|)$ where $|D|$ and $|T|$ are the sizes of the DTD and of the document, respectively. The complexity analysis is however very brief and it seems unclear how far it takes the recursion issues into account (the correction of the same node with respect to the same automaton might be necessary many times within one tree). The space complexity announced is of $O(|D|^2 \times height(T))$ since only two consecutive columns of the restoration graph need to be stored at a time. It is mentioned that restoring the correcting edit sequences from the restoration graph is straightforward as each sequence stems from one of the minimum-length graph paths. However, no hints are given on which paths are to be taken into account (all or just one, and which one). Moreover, no considerations are present concerning the position updates in the modified tree. The experimental results on randomly generated invalid XML documents show a linear behavior of the algorithm in function of the document size (with a DTD of size 5) and a polynomial behavior in function of the DTD size.

In [23] the authors reconsider the problem of the document-to-DTD edit distance in that they represent a DTD by a *streaming tree automaton* (STA), i.e. a single push-down automaton which operates on an input XML document seen as a sequence of opening and closing tags (in the standard document order) rather than as a tree. Given the STA for a DTD a *repair automaton* is defined which resembles again a matrix whose: (i) number of rows is equal to the number of states in the STA, (ii) number of columns is equal to the number of opening and closing tags in the XML document. Transitions in the repair automaton are weighted and represent edit operations (and their costs): (i) renaming a node, (ii) inserting a node (not necessarily a leaf), (iii) deleting a node (not necessarily a leaf). Deleting or inserting a root is not allowed. Finding the document-to-DTD distance is, here again, reduced to finding the minimal-weight accepting run in the repair automaton. This approach seems promising in that: (i) it considers a rich set of edit operations (insertions and deletions of internal nodes are allowed), (ii) the well-formedness as well as the attribute validity seem to be handled within the same framework as the structural validity. The algorithm is described within a more complex problem of querying sets of (possibly inconsistent) XML documents, proved EXPTIME-complete. The time complexity is not given explicitly for the problem of the document-to-DTD edit distance alone. We regret that no experimental results accompany this proposal, in particular with respect to large documents and complex DTDs.

In [2] the authors also consider streaming XML, thus they also propose to represent a schema with a push-down finite state machine called Visibly Pushdown Automaton (VPA), considering XML documents as strings with opening and closing symbols. A VPA can represent extended DTDs (EDTD, also called specialized DTDs), that define regular tree languages. The authors introduce Visibly Pushdown Transducers (VPT), that take as input a document and produce as output a document or a set of documents. They present first a VPT Γ designed as a way of computing an extension E of a given schema S, such that $L(E)$ contains $L(S)$ plus all documents whose edit distance from $L(S)$ is up to K. VPT Γ is a combination of all possible transformations consisting of up to K edit operations. The considered edit operations are the same as the ones described in [23]. Both S and E are represented by VPA, say A and B. VPA B is obtained by transducing A by Γ. It can be stored and used to test whether an XML document would fit the schema S after at most K edit operations. This test consists in verifying if the document is accepted or not by B, thus its time complexity depends linearly on the size of the document, and necessarily it depends also on features of B, which is non deterministic and which has $O(KM)$ states and $O(KR^2M)$ transitions, where R is the size of the underlying alphabet, K is the upper bound of the number of edit operations and M is the size of A. $O(KR^2M)$ is also the state and time complexity of the processing of B. The authors of [2] also extend their proposal to another VPT that represents changes expressed with special edit operations: substitution of all occurrences of pair $(<a>, </a>)$ (for instance) by pairs $(<b>, </b>)$, deletion of all occurrences of pair $(<a>, </a>)$ and insertion any number of pairs $(<a>, </a>)$. For these operations, the time complexity of processing B is $O(K^3R^{2K+1}M)$. The article ends with hints for a VPT that can generate "repairs" of a document that is already valid w.r.t. the original schema S.

6.2. Finding One Minimal-Cost Correction, or All Minimal-Cost Corrections

The reference [27] is frequently cited as it was probably the first to announce the following challenge: given a tree T and a tree grammar G such that $T \notin L(G)$, $L(G)$ being the tree language defined by G, find another tree T' that is in $L(G)$ and not too far from T. As in many other frameworks, trees stand for XML documents and the grammar is a DTD. Nevertheless in this paper the trees are presented as being ranked, which is not the case for XML document trees, so one can guess that the algorithms work on some binary representation of trees[10], that is not presented. The following elementary edit operations are considered: *relabel* of a node, *insertion* and *deletion* of a node at any level of the tree. The authors mention that one of them has proved in another article the existence of a "tester" for tree edit distance with a special edit operation called *move*, which allows the movement of a whole subtree from one node to another node in the tree. This paper focuses on experimental results performed for an implementation of (previously published) algorithms, which are unclearly sketched. These algorithms deal with classical tree edit distance (without moves). The first one consists in marking nodes that are parents of an error during a bottom-up traversal of the tree, then going top-down for "modifying the neighborhood" of each marked node. How these modifications are performed is not presented. The method is said to be linear in the size of the tree and exponential in the number of errors. The aim of the second algorithm is to avoid the exponential factor but it outputs a T' that may be not optimal. It is said that the key function in this second algorithm is to compute the distance between a string and a regular expression but no hints are given for doing that (there is a reference to a paper) and it does not seem to be taken into account in the complexity of the whole algorithm (but it is said that, for optimizing this computation, it is necessary to avoid some very common and useful forms of DTDs). The reported experiments concern only the second algorithm, for 4 different DTDs. For each DTD, 5 XML documents are automatically generated with sizes ranging from 50 nodes to 800 nodes and with always 10 errors, whose types and places are not specified. Execution times (for finding T') range from 900 to 2400 milliseconds. A web site is given for testing this implementation, which is no longer maintained.

In [28] and [29] the problem of correcting an XML tree with respect to a schema is based on our own definitions from [10] and expands them so as to: (i) deal with single type tree grammars, i.e. XML schemas, not only with DTDs, (ii) integrate the correction of attributes, not only elements. The algorithm adapts also the idea from [4] of modeling corrections by finding

[10]This is probably why the implementation uses SAX in order to build DOM trees.

minimal length paths in a correction graph, however not only does it find the distance between the tree and the schema but also restores the edit sequences necessary to obtain the corrected trees. It also uses the idea of caching previously corrected partial results, as in our approach. The authors mention experimental results whose time consumption shows a linear behavior in function of the document size. However the nature of the data and the conditions of the experiment are not described[11]. It is also suggested that finding all possible solutions within a given threshold, not only the minimal ones (which would make this framework even closer to ours), is possible after some modifications of the original algorithm. These are however not defined.

6.3. Computing a Set of Edit Operation Sequences for Correcting XML Documents w.r.t. a Schema

Apart from [28] and [29] mentioned above, the proposal in [3] is the only one we know that computes a set of edit scripts (i.e. operation sequences) between an XML document t and a tree grammar (attributes are not considered). It does not use a threshold th, but a number K of desired optimum corrections, i.e. it searches for K least-cost corrections. The first result of the paper is the proof that this problem is NP-hard[12]. The second result is a pseudopolynomial time algorithm for one-unambiguous XML grammars. The three edit operations considered, different from ours, are as follows: $ren(n_i, a)$ assigns the label a to the node n_i, $add(a, n_h, n_j)$ adds a node $n_{h,j}$ labeled by a *as the parent of siblings* $n_h, ;n_j$, and $del(n_i)$ deletes an internal node n_i (*its children take its place*). As usual, a cost is associated with each edit operation. To respect the semantics of these edit operations, the notion of edit script is defined which puts restrictions on the order of edit operations (moreover only a *ren* operation can be applied on the root and no leaf deletion is allowed). The author first defines a graph $H_K(t, N)$ containing $|t| + 1$ nodes and about $|N| \times K \times 2 \times |t|$ edges, N being the set of non-terminals in the grammar. $H_K(t, N)$ represents all changes that might occur on t. Some paths in this graph correspond to sequences of siblings, furthermore edges are associated with edit operations (with their cost). Then the author proposes to consider the subgraphs corresponding to all children of the same node n in order to compute *the intersections* between these subgraphs and the NFA that represents production rules associated with n's label. Finding the *k-th* optimum edit script consists in computing the *k-th* shortest path in the resulting graph. As it is possible to find K optimum edit-scripts for t provided that there exist known optimum edit-scripts for each proper subtree of t, the algorithm runs from lower to higher edges in $H_K(t, N)$, following a specified partial order.

Although it is not always relevant to provide K optimum edit scripts (how does the algorithm choose between two edit scripts having exactly the same cost?), the edit operations used in this proposal are interesting and the algorithm seems promising, considering that its complexity is said to be pseudo-polynomial. However there is no mention of any experiment in the paper and it is not possible to determine if there has been an implementation of its ideas since its publication.

6.4. Revalidation and Correction after Document or Schema Updates

Some particular instances of the tree-to-schema correction problem appear in the context of evolving XML documents and schemas, notably on the web.

On the one hand, *XML documents* previously known to be valid may be *subject to updates*, which may invalidate them. Two complementary types of approaches were proposed in order to face this problem:

- One tries to infer new schema constraints from the document updates, as in [15] or more recently in [16], where schema evolution is concerned within a streaming setting.
- One tries to correct the document in order to restore its validity, however priority is given to preserving the most recent updates. Our previous work in [9, 10] offers such a solution within an *incremental* framework. Namely, given a previously valid document t, on which a set of updates are performed, we target the correction process on the parts of the document concerned by the updates. During the incremental revalidation, a correction routine is activated whenever an invalid subtree is encountered. At the end of the revalidation, the corrections generated by each call to the routine are combined and several versions of valid documents are proposed to the user.

On the other hand, one may have to perform *updates on a schema*. As a result, previously valid documents may become invalid (although this is not always the case, as shown in [11]) and may require a correction with respect to the modified schema. In [12, 13] the author demonstrates that inferring K optimum transformations of an XML document from a DTD-update script is NP-hard (even for $K = 1$). He also presents a solution for the particular case of this problem, i.e. when the DTD-update script is of length 1. More precisely, given a DTD D, an elementary update u transforming D into D', and a document t valid w.r.t. D, the proposed algorithm computes the list of top-K transformations of t (inferred from u), each of which transforms t to a document which is valid with

[11]The implementation of this approach is downloadable but the testing data used in the reported experiments are not provided. For the sake of a comparison, we have tried to run the implementation on our own testing data, which resulted in a program crash. We noticed that the form of the schema is non standard. It seems that recursive definitions are not allowed.

[12]Finding a solution is exponential in the size of the inputs.

respect to D'. The algorithm is shown to be polynomial in the size of D, t and K. The problem of adapting previously valid documents after a modification of their schema is also addressed in [14]: the schema is an XML Schema (an XSD) and only one heuristically chosen transformation is computed. No complexity evaluation is given but experimental results demonstrate a linear time increase with respect to the document size.

The instances of the document-to-schema correction problem considered in this subsection are particular in the sense that documents are known to be initially valid before updates are applied on a document or on the schema. It is possible to take advantage of this knowledge as shown in [12, 13], [9, 10] and [30, 31]. Despite this fact [12, 13] shows that transforming a document into *a defined set* of valid documents (whether K-optimum or all within a given threshold th) is not a trivial task, even when it is known that the document was previously valid. One of the difficulties is precisely to state the properties of the computed solutions (w.r.t. the not computed ones). Similarly, in [9, 10] we can avoid revalidating and correcting substantial parts of the document but the set of the resulting solutions does not contain all valid documents within the threshold. Thus, it is impossible to know if the best solution w.r.t. the user's goals is contained in this set.

6.5. Contrastive Study

Table 1 shows a contrastive study of all cited approaches[13] with respect to how they define the problem of the tree-to-language correction. In most approaches this problem is expressed in terms of the tree-to-language edit distance. The distance measure most often builds upon the tree-to-tree distance, which in its turn relies on elementary edit operation on tree nodes. Only in [6, 7] no tree-to-tree distance is used and a direct mapping between the document and the schema is searched for.

Actual edit sequences and the resulting corrections for the invalid input tree are proposed in three approaches only: in [28, 29] all minimal corrections are output, [3] offers a fixed number of the closest solutions, and our own algorithm is the only one to propose a complete set of corrections within a given threshold[14]

All approaches deal with the structural validity of documents but three of them ([8], [23], [28, 29]) propose a more comprehensive framework in which the correctness of attributes is also accounted for, sometimes in a rather restricted framework (e.g. attributes of the same element are seen as sequences rather than sets).

[13]Except those of Section 6.4, which are hardly comparable with the general ones.

[14]As mentioned before, our tool actually also allows the output of only all minimal solutions, in which case no threshold is needed on input.

Different versions of the schema model are used in the literature. [27] and [8] use a particular restricted version of a DTD, where operators (?, +, $*$) are applied to elements only. Correspondingly, in [27] only ranked (binary) trees are dealt with, while all other works address unranked trees. Most other approaches seem to take a DTD without restriction (i.e. a local tree grammar) on input. As mentioned in [1], admitting recursive declarations within the schema is an important issue, however the relevant data are rarely explicitly available in the literature. Three approaches ([5], [3] and [28, 29]) allow a more general schema model: the XML Schema (i.e. single type tree grammar), and one approach ([2]) deals with the most general model of an extended (or specialized) DTD (i.e. a regular tree grammar).

There is an interesting correlation between how different approaches view the XML document, and which schema model and elementary operations they select.

- If the XML document is seen as a **tree**, the schema must obviously be a tree grammar (local or single-type), sometimes represented in a particular way, e.g. as a tree [8, 6, 7] or as a hedge grammar [5]. The well-formedness is not an issue here since an ill-formed document is not a tree. In this case (in [5], [4], [8], [28, 29] and in our proposal), the most natural elementary operations (except node relabeling) seem to be those concerning leaves rather than internal node. An exception to this rule is [3], and possibly also [27].
- The remaining approaches ([23] and [2]) view an XML document as a **word** of opening and closing tags and the schema is transformed to a pushdown automaton on words. This view offers a rather natural framework for well-formedness issues (e.g. correcting a missing closing tag comes down to inserting a character in a word). But most importantly, also elementary edit operations on internal tree nodes seem to be rather natural in this context.

Interestingly enough, all elementary operations mentioned in the state-of-the art are limited to node relabeling, inserting or deleting. Other potentially useful operations, proposed within the (simpler) problem of the tree-to-tree correction [24], are not yet addressed in the tree-to-language correction. Such new potential operations might include moving a whole subtree within a tree[15], inverting sibling subtrees[16], etc.

In Table 2, we consider again all cited articles within a contrastive study of their informativeness and reproducibility. We check if precise information is given

[15]This operation is defined in [27], but not addressed by their algorithms.

[16]Note that inversions of characters were considered as elementary operations from the very beginning of the word-to-word and word-to-language correction domain [20].

Reference	Elementary edit operations	Validity aspects			Algorithm's output					Schema type	Document model	Schema model
		Well-formedness	Structure	Attributes	Tree-to-schema edit distance	Corrections: minimal	Corrections: k closest	Corrections: all within a threshold	Edit sequences			
Boobna, de Rougemont [27]	node relabeling node insertion node deletion		✓			not always the minimal one				restricted DTD[a]	ranked ordered labeled tree	set of reg. exp.
Bertino et al. [6, 7]	no edit operation		✓		✓					DTD	unranked ordered labeled tree	ordered labeled tree
Xing et al. [5]	node relabeling leaf insertion leaf deletion		✓		✓					DTD and XML schema	unranked ordered labeled tree	regular hedge grammar
Staworko, Chomicki [4]	leaf insertion leaf deletion		✓		✓					DTD	unranked ordered labeled tree	top-down finite tree automaton
Suzuki [3]	node relabeling node insertion node deletion		✓		✓		✓		✓	DTD and XML schema	unranked ordered labeled tree	regular tree grammar
Tekli et al. [8]	node relabeling leaf insertion leaf deletion		✓	✓[b]	✓	✓				restricted DTD[c]	unranked ordered labeled tree	ordered labeled tree
Staworko et al. [23]	node relabeling node insertion node deletion (except a root)	✓	✓	✓	✓					DTD	serialized unranked ordered labeled tree	streaming tree automaton
Thomo et al. [2]	node relabeling node insertion node deletion (+ multi operations)	✓	✓		✓ not exactly			no but could serve to do it		Extended DTD	serialized unranked ordered labeled tree[d]	visibly pushdown automaton
Svoboda, Mlýnková [28, 29]	node relabeling leaf insertion leaf deletion		✓	✓	✓	✓			✓	XML schema	unranked ordered labeled tree	top-down finite tree automaton
Ours	node relabeling leaf insertion leaf deletion		✓		✓			✓	✓	DTD	unranked ordered labeled tree	set of reg. exp.

TABLE 1. Components of problem definition in tree-to-language correction approaches

[a]Called a unary normal form DTD.
[b]The treatment of attributes is limited in this approach. Moreover, attributes of the same element are seen as sequences rather than sets.
[c]Called a disjunctive normal form DTD.
[d]Called XML formatted word

Reference	Complexity estimation[a]		Proofs				Nature of data used in experiments	Availability				
	Time	Space	Correctness	Completeness	Termination	Complexity		Executable	Source code	License	Benchmark data	Web page
Boobna, de Rougemont [27]	$O(\|t\|)$						synthetic data, up to 800 nodes					[b]
Bertino et al. [6, 7]	$O((\|t\| + \|S\|) \times f_t^2)$						synthetic and real life data 11.111 nodes					
Xing et al. [5]	$O(\|t\| \times \|S\| \times \log \|S\|)$											
Staworko, Chomicki [4]	$O(\|S\|^2 \times \|t\|)$	$O(\|S\|^2 \times h(t))$					synthetic data 50 nodes	✓	✓	unknown		[c]
Suzuki [3]	$O(k \times \|\Sigma\| \times \|t\|^2 \times \|S\| \times R + k \times logk$		✓									
Tekli et al. [8]	$O((max(\|t\|, \|S\|)^3)$						synthetic and real-life data for classifying					
Staworko et al. [23]						[d]		✓	✓	unknown		[e]
Thomo et al. [2]	$O(th \times \|S\| \times \|\Sigma\|^2)^f$	$O(th \times \|S\| \times \|\Sigma\|^2)^g$	✓	✓		✓						
Svoboda, Mlýnková [28, 29]							synthetic data, 10,000 nodes	✓	✓	unknown		[h]
Ours	$O((f_t + 1) \times (f_S)^{\|t\|+th} \times 6 \times \|\Sigma\| \times (\|t\| + th))^{th}$		✓	✓	✓	✓	real-life data, 450 tree nodes th=0,...,16	✓	✓	LGPL v3	✓	[i]

TABLE 2. Properties and results of the tree-to-language correction approaches

[a] $|\Sigma|$ = size of the alphabet, $|t|$ = size of the tree, f_t = maximum fan-out of the tree, $h(t)$ = height of the tree, $|S|$ size of the schema, f_S = maximum fan-out of the schema, th = threshold, k = number of computed valid documents, R = maximum size of regular expressions in S

[b] http://www.lri.fr/∼mdr/xml.This demo does no longer produce any output.

[c] http://researchers.lille.inria.fr/ staworko/research/rhino-0.1.zip. It is unclear if this page contains the software described by this bibliographical reference.

[d] Proofs of exponential complexity are provided for a broader problem – the one of consistent querying of XML documents.

[e] http://researchers.lille.inria.fr/ staworko/research/hippo/index.html. It is unclear if this page contains the software described by this bibliographical reference.

[f] This is the time for building a VPA that recognizes all documents that are up to th far from a schema S.

[g] This is the space for storing the VPA.

[h] http://www.ksi.mff.cuni.cz/ svoboda/projects/corrector/downloads.php

[i] http://codex.saclay.inria.fr/deliverables.php

on their complexity, if necessary proofs are provided, and if implementations and testing data are available.

It appears, in view of this study, that the time complexity announced in the corresponding papers varies widely: from linear ([27] and [5]), through polynomial ([6, 7], [4], [3], [8], and [2]), through exponential (ours). Some approaches ([23] and [28, 29]) lack complexity estimation. It seems that the complexity heavily depends on the problem definition, in particular on the fact that edit sequences and the corresponding corrections are generated or not, and that the correction set is complete. More precisely, the complexity is estimated for two out of three approaches where edit sequences are computed:

- [3] announces a polynomial complexity but its correction set is bounded (the k closest corrections are found).
- Our approach has an exponential complexity but it is complete (all corrections within a threshold are found).

Only two references ([4] and [2]) provide an estimation of space complexity, only three of them ([3], [2] and ours) prove the algorithm's correctness, only two of them ([2] and ours) prove its completeness and complexity, and only one (ours) proves its termination. In one approach [23] proofs are provided for a broader problem only (consistent querying of XML documents).

Experimental results are provided by six approaches ([27], [6, 7], [4], [8], [28, 29] and ours). A few of them (including ours) operate on real-life rather than synthetic data. The sizes of corrected documents vary from 50 to over 11,000 nodes.

Four approaches offer downloadable implementations (executables and/or source codes). Two of them ([4] and [23]) lack any user's documentation, thus it is unclear how to run them and if they really address the tree-to-language correction problem. Another one ([28, 29]), which admits a non standard schema format, was tested with the DTD in Fig. 20 and worked only after the schema recursion has been removed. Our approach seems to be the only one that offers, in addition to the executable and the source code, also the user's guide and the set of testing data used to obtain the experimental results. Consequently, it seems to be the only reproducible one. Last but not least, our source code is the only one to be distributed under a known license, namely the open license GNU LGPL v3.

In view of the above analysis, we think that our proposal brings a substantial contribution to the field of tree-to-language correction. We claim that, relatively to the state of the art presented, we offer the first full-fledged solution to this problem. Obviously, many extensions and enhancements of our algorithm and implementation are still possible. Some of them have been discussed in Section 4.4.

7. CONCLUSION

Tree-to-language correction is a theoretical problem which has a number of interesting existing applications in the field of XML processing, and certainly many future applications.

We have presented an algorithm for, given a well-formed XML document seen as a tree t, a DTD seen as a schema structure S, and a non negative threshold th, computing every tree t' valid with respect to S such that the edit distance between t and t' is no higher than th. The edit distance between trees, inspired by [17], is based on three elementary node edit operations: (i) relabeling a node, (ii) adding a leaf, (iii) deleting a leaf. These operations can be grouped to more complex tree edit operations: (i) inserting a subtree, (ii) removing a subtree. The schema is represented as a set of finite-state automata associated with labels. The algorithm extends the ideas from [18] in that the FSA of the current node's label is traversed in the th-bound depth-first order and an edit distance matrix is completed column by column each time a new transition is followed. However since we correct trees and not only strings, following each transition may potentially provoke a recursive correction of a subtree of the current node. Moreover the edit distance matrix stores the relevant edit operation sequences allowing us to obtain the correction tree candidates.

We have proved the correctness and the completeness of the algorithm. We have shown that its worst-case time complexity is $O((f_t + 1) \times (f_S)^{|t|+th} \times (6 \times |\Sigma| \times (|t| + th))^{th}$, where f_t is the maximum fan-out in t, f_S is the maximum fan-out of all FSAs in S, $|t|$ is the number of t's nodes, Σ is the alphabet of S, and th is the correction threshold. This theoretical exponential complexity is related to the fact that edit sequences and the corresponding corrections are generated and that the correction set is complete.

We have performed experimental tests on real-life data focused on the influence of different input parameters (t's size, th, number of errors in t, position of an error in t) on the correction time and on the number of solutions found. In particular, the experimental CPU time consumption shows a polynomial rather than an exponential behavior.

In the light of the detailed related work analysis, the main contribution of our approach is to offer the first full-fledged study of the problem of the document-to-schema correction problem. Not only do we measure the distance between a document and a schema but also find the candidate correction trees. We do not limit ourselves to finding the minimal solution but find all solutions within a threshold instead. Thus, we consider the correction as an enumeration problem rather than a decision problem, contrary to most other approaches.

Some recent approaches such as [3], [23] and [29] shed new light on the document-to-schema correction problem in that they introduce edit operations acting on internal nodes and offer optimizations of data structures via graph-based modeling. One of our perspectives is to examine how these proposals can be integrated with ours so as to propose a more universal framework in which different variants of the correction problem might be solved most efficiently.

8. FUNDING

This work was supported by the French Research Agency (ANR) [ANR-08-DEFIS-04], for the CODEX project: *Efficiency, Dynamicity and Composition for XML – Models, Algorithms and Systems*, 2008–2012.

9. ACKNOWLEDGEMENTS

We are deeply indebted to the three anonymous referees for their thorough reviews of our paper, which greatly enhanced its quality and readability.

REFERENCES

[1] Tekli, J., Chbeir, R., Traina, A., and Traina, C. (2011) XML document-grammar comparison: related problems and applications. *Central European Journal of Computer Science*, **1**, 117–136.

[2] Thomo, A., Venkatesh, S., and Ye, Y. Y. (2008) Visibly Pushdown Transducers for Approximate Validation of Streaming XML. *Proceedings of FoIKS 08, Pisa, Italy*, 11–15 February, Lecture Notes in Computer Science, **4932**, pp. 219–238. Springer.

[3] Suzuki, N. (2007) Finding K Optimum Edit Scripts between an XML Document and a RegularTree Grammar. *Proceedings of EROW 07, Barcelona, Spain*, 13 January. CEUR-WS.org.

[4] Staworko, S. and Chomicki, J. (2006) Validity-Sensitive Querying of XML Databases. *Proceedings of EDBT 06, Munich, Germany, Revised Selected Papers*, 26–31 March, Lecture Notes in Computer Science, **4254**, pp. 164–177. Springer.

[5] Xing, G., Malla, C. R., Xia, Z., and Venkata, S. D. (2006) Computing Edit Distances Between an XML Document and a Schema and its Application in Document Classification. *Proceedings of SAC 06, Dijon, France*, 23–27 April, pp. 831–835. ACM.

[6] Bertino, E., Guerrini, G., and Mesiti, M. (2004) A Matching algorithm for measuring the structural similarity between an XML documents and a DTD and its applications. *Information Systems*, **29**, 23–46.

[7] Bertino, E., Guerrini, G., and Mesiti, M. (2008) Measuring the structural similarity among XML documents and DTDs. *Journal of Intelligent Information Systems*, **30**, 55–92.

[8] Tekli, J., Chbeir, R., and Yétongnon, K. (2007) Structural Similarity Evaluation Between XML Documents and DTDs. *Proceedings of WISE 07, Nancy, France*, 3–7 December, pp. 196–211.

[9] Bouchou, B., Cheriat, A., Halfeld Ferrari Alves, M., and Savary, A. (2006) XML Document Correction: Incremental Approach Activated by Schema Validation. *Proceedings of IDEAS 06, Delhi, India*, 11–14 December, pp. 228–238. IEEE Computer Society.

[10] Bouchou, B., Cheriat, A., Halfeld Ferrari Alves, M., and Savary, A. (2006) Integrating Correction into Incremental Validation. *Proceeding of BDA 06, Lille, France*, 17–20 October.

[11] Bouchou, B. and Duarte, D. (2007) Assisting XML Schema Evolution that Preserves Validity. *Proceedings of SBBD 07, João Pessoa, Paraíba, Brasil*, 15–19 October, pp. 270–284. SBC.

[12] Suzuki, N. (2008) On Inferring K Optimum Transformations of XML Document from Update Script to DTD. *Proceedings of COMAD 08, Mumbai, India*, 17-19 December, pp. 210–221. Computer Society of India / Allied Publishers.

[13] Suzuki, N. (2010) An algorithm for inferring *k* optimum transformations of xml document from update script to dtd. *IEICE Transactions*, **93-D**, 2198–2212.

[14] Guerrini, G., Mesiti, M., and Sorrenti, M. (2007) XML Schema Evolution: Incremental Validation and Efficient Document Adaptation. *Proceedings of XSym 07, Vienna, Austria*, 23–24 September, pp. 92–106. Springer.

[15] Bouchou, B., Duarte, D., Alves, M. H. F., Laurent, D., and Musicante, M. A. (2004) Schema Evolution for XML: A Consistency-Preserving Approach. *Proceedings of MFCS 04, Prague, Czech Republic*, 22–27 August, Lecture Notes in Computer Science, **3153**, pp. 876–888. Springer.

[16] Shoaran, M. and Thomo, A. (2011) Evolving schemas for streaming XML. *Theoretical Computer Science*, **412**, 4545–4557.

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

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

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

[20] Boytsov, L. (2011) Indexing methods for approximate dictionary searching: Comparative analysis. *ACM Journal of Experimental Algorithmics*, **16**.

[21] Du., M. W. and Chang, S. C. (1992) A model and a fast algorithm for multiple errors spelling correction. *Acta Informatica*, **29**, 281–302.

[22] Murata, M., Lee, D., Mani, M., and Kawaguchi, K. (2005) Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Technol.*, **5**, 660–704.

[23] Staworko, S., Filiot, E., and Chomicki, J. (2008) Querying Regular Sets of XML Documents. *Proceedings of LiD 08, Rome, Italy.*

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

[25] Bille, P. (2005) A survey on tree edit distance and related problems. *Theoretical Computer Science*, **337**, 217–239.

[26] Savary, A., Waszczuk, J., and Przepiórkowski, A. (2010) Towards the Annotation of Named Entities in the Polish National Corpus. *Proceedings of LREC 10, Valletta, Malta*, 17-23 May. European Language Resources Association.

[27] Boobna, U. and de Rougemont, M. (2004) Correctors for XML Data. *Proceedings of XSym 04, Toronto, Canada*, 29–30 August, Lecture Notes in Computer Science, **3186**, pp. 97–111. Springer.

[28] Svoboda, M. (2010) Processing of Incorrect XML Data. Master's thesis. Charles University in Prague.

[29] Svoboda, M. and Mlýnková, I. (2011) Correction of Invalid XML Documents with Respect to Single Type Tree Grammars. *Proceedings of NDT 11, Macau, China*, 11–13 July, Communications in Computer and Information Science, **136**, pp. 179–194. Springer.

[30] Raghavachari, M. and Shmueli, O. (2004) Efficient Schema-Based Revalidation of XML. *Proceedings of EDBT 04, Heraklion, Crete, Greece*, 14–18 March, pp. 639–657.

[31] Raghavachari, M. and Shmueli, O. (2007) Efficient Revalidation of XML Documents. *IEEE Trans. Knowl. Data Eng.*, **19**, 554–567.

APPENDIX A. OPTIMIZATION STEP

Our main optimization consists in saving the intermediate correction results in an auxiliary structure in such a way that the correction of a subtree of t for the target root label a and with the threshold th is performed at most once. Consequently, if the correction of a subtree t with a threshold th_1 is needed, and the correction for the same subtree with a bigger threshold th has already been computed, the needed computation will not be performed and the result will be the stored sequences having a cost less or equal to th_1. The auxiliary structure is a table $ResultTable$ whose items are of type *MResult*, defined as follows:

```
struct MResult {
    t: XML tree
    th: natural (threshold)
    c: character (root tag of resulting trees)
    Result: set of node-edit operation sequences
}
```

The type *MResult* involves the corrected tree, the root tag of the resulting trees, the threshold for which the corrections have been calculated, and the node-edit operation sequences representing the corrections.

The auxiliary structure $ResultTable$ is implemented with a Hashtable and used as a global variable. It is managed by two main methods:

- *saveResult*: for saving the result of a correction.
- *getResult*: for retrieving the result of a correction. This function returns $\emptyset$, $false$) if no correction has been stored yet. It returns (*Result*, *true*) if the corresponding correction has already been performed.

At the beginning of the function *correction* below, the auxiliary structure is accessed via the *getResult* function (line 2). Thus, if the current subtree t has already been corrected with respect to the same root label c and with a threshold no lower than th, the corrections with distance no higher than th are retrieved and do not have to be recalculated (line 3). Otherwise, the correction is performed and the result is stored in the auxiliary structure via the *saveResult* procedure (line 22), before being returned (line 23).

```
Procedure saveResult($t$, $th$, $c$, $Result$)
Input
$t$: XML tree ; $th$: natural (threshold)
$c$: character (root tag) ; $Result$: set of node-edit operation sequences
1.  begin
2.     $R := ResultTable.getElement(t, c)$
3.     if $R \neq null$ then
4.        if $R.th < th$ then
5.           $R.th := th$
6.           $R.Result := Result$
7.        end if
8.     else
9.        $R' := new\ MResult(t, th, c, Result)$
10.       $ResultTable.add(R')$
11.    end if
12. end
```

```
Function getResult($t$, $th$, $c$) return ($Result$,$b$)
Input
$t$: XML tree ; $th$: natural (threshold) ; $c$: character (root tag)
Output
$Result$: set of node-edit operation sequences
$b$: boolean ($false$ if no Result exists yet, otherwise $true$)
1.  begin
2.     $R := ResultTable.getElement(t, c)$
3.     if $R \neq null$ and $th \leq R.th$ then
          //Function getResultMaxCost returns a set of node-edit
          //operation sequences having cost no higher than th
4.        return ($getResultMaxCost(R.Result, th)$, $true$)
5.     return ($\emptyset$, $false$)
6.  end
```

```
Function correction($t$, $S$, $th$, $c$) return $Result$
Input
$t$: XML tree (to be corrected) ; $S$: structure description
$th$: natural (threshold) ; $c$: character (intended root tag of resulting trees)
Output
$Result$: set of node-edit operation sequences (allowing to get resulting trees)
Local variable
$b$: boolean (just for optimization)
1.  begin
2.     $(Result, b) := getResult(t, th, c)$
3.     if $b \neq false$ then return $Result$
4.     else if $th = 0$ and $t \in L_{loc}(S)$ and $t(\epsilon) = c$ then
5.        return $\{nos_\emptyset\}$ //Stop recursion
6.     else if $th \leq 0$ then
7.           return $\emptyset$ //Stop recursion
8.        else
9.           $u := \epsilon$
10.          $n := \bar{t}$ //$n$ is the number of t's root's children
11.          $M_u^c := newMatrix(n{+}1, 1)$ //Initialize the matrix with $n+1$ rows and 1 column
             //Compute the first column in the matrix.
12.          if $t = t_\emptyset$ then
13.             $M_u^c[0][0] := \{(add, \epsilon, c)\}$
14.          else
15.             if $c = t(\epsilon)$ then
16.                $M_u^c[0][0] := \{nos_\emptyset\}$
17.             else
18.                $M_u^c[0][0] := \{(relabel, \epsilon, c)\}$
19.          for $i := 1$ to $n$ do
20.             $M_u^c[i][0] := \{(remove, i{-}1, t_\emptyset)\}._{th} M_u^c[i{-}1][0]$
21.          $correctionState(t, S, th, c, M_u^c, initialState(FSA_c), Result)$
22.          $saveResult(t, th, c, Result)$
23.          return $Result$
24. end
```