PAPER Special Section on Formal Approaches

Direct Update of XML Documents with Data Values Compressed by Tree Grammars

Kenji HASHIMOTO[†a)], ***Member***, **Ryunosuke TAKAYAMA**[†], ***Nonmember***, ***and*** **Hiroyuki SEKI**[†b)], ***Fellow***

SUMMARY One of the most promising compression methods for XML documents is the one that translates a given document to a tree grammar that generates it. A feature of this compression is that the internal structures are kept in production rules of the grammar. This enables us to directly manipulate the tree structure without decompression. However, previous studies assume that a given XML document does not have data values because they focus on direct retrieval and manipulation of the tree structure. This paper proposes a direct update method for XML documents with data values and shows the effectiveness of the proposed method based on experiments conducted on our implemented tool.

key words: *XML, compression, tree grammars, update*

1. Introduction

Digital documents that have been generated, communicated and stored in computer systems are rapidly increased and the development of efficient document compression methods are required. Many of those documents are described by XML (Extensible Markup Language), which is a *de facto* standard of a markup language for structured documents. Since an XML document is given an internal hierarchical structure by arbitrarily nested tags, the document can be modeled by an (unranked) tree. By this reason, a few compression methods for trees have been proposed. One of the most promising compression methods is the one that translates a given tree t to a tree grammar $\mathcal{G}$ that generates (only) t. The idea is that we replace frequent occurrences of a common substructure s with a nonterminal symbol, say A, and instead, we introduce a production rule $A \to s$. TreeRePair [1] is a compression tool that achieves efficient processing and good compression ratio simultaneously by repeatedly replacing frequently occurring tree digrams with a nonterminal symbol of a straight-line context-free tree grammar (SLCFTG). Note that a dag (directed acyclic graph) is a familiar compression of a tree, and it is regarded as a straight-line regular grammar, which forms a proper subclass of SLCFTG.

When we make access to a compressed data c of an original tree t, we decompress c to reconstruct t, retrieve t, and recompress t' if we update t to t'. It is desirable to directly manipulate c without decompression. In compression methods based on tree grammars, such a direct manipulation is possible because the internal structures are kept in production rules of the grammar. Along this line, some direct manipulation tools have been already reported [2]–[8]. However, previous studies on compression and direct manipulation assume that a given XML document does not have data values such as attribute values and PCDATA because those studies focus on investigating an efficient compression and direct retrieval of the structure.

This paper proposes a direct update method for structured documents with data values and shows the effectiveness of the proposed method based on experiments conducted on our implemented tool. Our approach is as follows. A structured document with data values is modeled by a data tree (t, δ) where t is an ordinary tree and δ is a valuation function that takes a node of t and an attribute name and returns an associated data value. We assume t is compressed into an SLCFTG $\mathcal{G}$ by TreeRePair to obtain a compressed data tree $(\mathcal{G}, \delta)$. Note that compression of a valuation function δ is out of the scope of this paper. In a real setting, the data part could be compressed by a traditional method independently.

An update instruction is specified as $(\mathcal{A}, op)$ where $\mathcal{A}$ is a query that selects the positions in the document that should be updated and op is a kind of updated operation possibly with arguments, such as relabel, delete, insert and update-value. As a formal model of a query $\mathcal{A}$, we introduce a deterministic selecting top-down data tree automaton with bottom-up look-ahead. A transition rule of the look-ahead part is conditional in the sense that it determines the state assigned to the current node depending on the truth value of a specified predicate on the data value of a specified attribute associated with the current node as well as the label of the current node and the states assigned to the children nodes. The look-ahead part is useful to simulate filters appearing in XPath.

An advantage of direct update without decompression is that we can avoid duplicate query evaluation on common substructures. For this purpose, the proposed query evaluation algorithm uses memoization, i.e., keeping the evaluated partial runs of a given automaton in a hash map. Note that the algorithm can reuse the run stored in the map when not only the substructure but also all the decisions made by the applied conditional transition rules are the same as before because the data values may be different from the previous ones. As another implementation-level optimization, we divide a file that stores a valuation function δ into a few files.

Manuscript received August 21, 2017.
Manuscript revised January 6, 2018.
Manuscript publicized March 16, 2018.

[†]The authors are with the Graduate School of Informatics, Nagoya University, Nagoya-shi, 464–8601 Japan.
a) E-mail: k-hasimt@i.nagoya-u.ac.jp
b) E-mail: seki@i.nagoya-u.ac.jp

DOI: 10.1587/transinf.2017FOP0002

When the positions to be updated are not distributed over the tree, this optimization can prevent the algorithm from loading the files not related to the updated positions. We conduct experiments of query evaluation and updates with some benchmark XML documents and the results show that our method outperforms BaseX [9] in both of CPU time and memory usage.

Related work Compression of tree-structured data has been studied extensively since [10] proposed a compression method via DAGs (also see [11]). Using tree grammars improves the compression ratio because we can merge repeated sub-structures that are not always complete subtrees, and a few compression tools based on straight-line context-free tree grammars (SLCFTGs) have been developed such as BPLEX [12], Clux [13] and TreeRePair [1] (Note that compression via DAGs can be regarded as compression via regular tree grammars, i.e., CFTGs where no nonterminal symbol has an argument.) Among others, TreeRePair shows good compression ratio as well as efficiency in compression and this study has used TreeRePair as the basic compression tool. The problem of constructing a minimal SLCFTG is computationally intractable. [14] proposed a recompression algorithm for string grammars that replaces (string) digrams in linear time, and it is extended in [15] to a compression algorithm for SLCFTGs, which exhibits the best proven approximation ratio w.r.t. the minimal SLCFTG.

About direct manipulation of compressed structured data, [4] proposed a method of updating compressed data without full decompression, called *path isolation*. Their method traverses grammar rules (compressed data), expanding (decompressing) non-terminals by the corresponding rules until an update position is reached, conducts an update and recompresses the updated tree locally. Path isolation has been extended in various ways. [5] proposed a parallelization of path isolation in updating grammars. [2] further optimized an updating algorithm by representing queries by DAGs and [6] proposed an extension of SLCFTG so that redundancies appearing in path isolation can be reduced. In [7], a recompression method is proposed such that tree digrams across more than one rules are searched in an SLCFTG updated by path isolation, and the detected digrams are locally compressed by TreeRePair. The experimental results of [7] showed that their recompression method achieved almost the same compression ratio as the one given by decompression, update and recompression. Our proposed method takes a copy of the rule that directly or indirectly contains an update position and when an update position is found, an update operation is conducted. To improve the compression ratio after an update, our method keeps track of the update history and eliminates the created copy if there already exists a copy of the rule with same history, and the method also conducts a (weak version of) pruning [1] after an update. All the update methods except ours do not deal explicitly with compressed documents with data values.

As a theoretical study, [16] discusses the computational complexity of querying trees compressed by tree grammars.

Top Tree is another data structure that can represent a tree in a compact way [17]. An updating method similar to this paper for documents (without data values) compressed by top trees is reported in [8]. Though the sizes of compressed documents via top trees are experimentally about twice as large as those by TreeRePair (see [18]), basic information (e.g., reachability between specified node pairs) can be maintained efficiently through updates in top trees [19]. Recently, a compression method of graphs via graph grammars was proposed based on graph digrams [20]. Extending our method to graph compression is another interesting future work.

2. Preliminaries

2.1 Trees

A *ranked alphabet* Σ is a finite set of symbols with $\mathrm{rk} : \Sigma \to \mathbb{N}$. For a symbol $a \in \Sigma$, the value $\mathrm{rk}(a)$ is called the *rank* of a. Let Σ_n denote $\{a \in \Sigma \mid \mathrm{rk}(a) = n\}$. Let Att be a set of *attribute names*.

Definition 1: A *labeled ordered tree over* Σ (simply called a *tree*) is a pair (D, λ) such that

- $D \subseteq \mathbb{N}_+^*$ where $\mathbb{N}_+ = \{1, 2, 3, \ldots\}$,
- λ is a total function from D to Σ,
- if $p \cdot p' \in D$ then $p \in D$, and
- if $p \in D$ and $\lambda(p) \in \Sigma_n$, $p \cdot i \in D$ for each i $(1 \leqq i \leqq n)$.

For a tree $t = (D, \lambda)$, we call an element in D a *node* (or a *position*) of t. We call λ the *labeling* function of t. For nodes $p, p \cdot i \in D$ where $p \in \mathbb{N}_+^*, i \in \mathbb{N}_+$, $p \cdot i$ is the ith *child* of p and p is the *parent* of $p \cdot i$. For nodes $p, p' \in D$, if $\exists u \in \mathbb{N}_+^*$. $p = p' \cdot u$, denoted by $p \geq p'$, then p is a *descendant* of p', and p' is an *ancestor* of p. A *leaf* is a node having no child, and the *root* is ϵ. A *subtree* of t rooted by $p \in D$, denoted by $t|_p$, is the tree (D_p, λ_p) such that $D_p = \{p' \mid p \cdot p' \in D\}$ and $\lambda_p(p') = \lambda(p \cdot p')$ if $p \cdot p' \in D$.

Definition 2: A *labeled ordered data tree over* (Σ, Att) (simply called a *data tree*) is a pair (t, δ) such that

- $t = (D, \lambda)$ is a tree over Σ, and
- δ is a partial function from $D \times Att$ to $\mathbb{N}$.

For a data tree $\tau = ((D, \lambda), \delta)$, we call δ the *valuation* function of τ. For $p \in D$ and $\alpha \in Att$, if $\delta(p, \alpha)$ is defined and $\delta(p, \alpha) = d \in \mathbb{N}$, we say that the data value on attribute α at p is d. We call the tree (D, λ), denoted by $\mathrm{st}(\tau)$, the *structure* of τ. A *subtree* $\tau|_p$ of τ rooted by $p \in D$ is the data tree $(\mathrm{st}(\tau)|_p, \delta_p)$ such that the domain E of δ_p is $\{(p', \alpha) \mid \delta(p \cdot p', \alpha) \text{ is defined}\}$ and $\delta_p(p', \alpha) = \delta(p \cdot p', \alpha)$ for each $(p', \alpha) \in E$.

The above trees are *ranked*, i.e., the number of children of nodes are determined by their labels. XML documents are modeled by *unranked* labeled ordered trees. Every unranked tree can be encoded to a ranked tree by the first-child next-sibling (fcns) encoding [21]. In what follows, we give a definition of fcns encoding, as a mapping from sequences

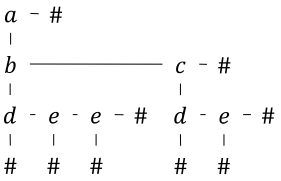


Fig. 1 Structure (D, λ)

Table 1 Valuation function δ

position	attr	value
11	x	1
112	y	10
112	z	100
1122	y	11
121	x	2
1212	y	20

of unranked trees over Σ to binary trees over $\Sigma \cup \{\#\}$, where $\#$ is a symbol such that $\mathrm{rk}(\#) = 0$ and $\# \notin \Sigma$.

Definition 3: Let $t_1, \ldots, t_n$ be unranked labeled ordered trees, and $t_1 = f(s_1, \ldots, s_m)$. $\mathrm{fcns}(t_1 \cdots t_n)$ is defined inductively as follows:

- $\mathrm{fcns}() = \#$, and
- $\mathrm{fcns}(t_1 \cdots t_n) = f(\mathrm{fcns}(s_1 \cdots s_m), \mathrm{fcns}(t_2 \cdots t_n))$.

In trees obtained by the fcns encoding, the rank of any symbol $f \in \Sigma$ is two. Note that each internal node of the obtained binary tree corresponds to a node of the unranked tree, and vice versa. The fcns encoding can be naturally extended to that from unranked data trees to binary data trees. Henceforth, we consider only binary data trees obtained by the fcns encoding as an input tree before compression. Note that we use trees which is not binary in tree grammars, or compressed trees, we will introduce later.

Example 1: Consider a data tree with the structure (D, λ) shown in Fig. 1 and the valuation function δ shown in Table 1. In the data tree, the label of 112 is e, and the data values on attributes y and z at 112 are 10 and 100, respectively.

2.2 Straight-Line Context-Free Tree Grammars

Let $\mathcal{X} = \{x_1, x_2, \ldots\}$ be a countable set of variables disjoint with Σ. We denote by $T(\Sigma, \mathcal{X})$ the set of all trees over $\Sigma \cup \mathcal{X}$ where the rank of any variable is 0. A tree $t \in T(\Sigma, \mathcal{X})$ is *linear* if any variable appears at most once in t. For $t, t_1, \ldots, t_n \in T(\Sigma, \mathcal{X})$, $t[x_1/t_1, \ldots, x_n/t_n]$ denotes the tree obtained from t by replacing x_i with t_i $(1 \le i \le n)$.

Let $z \notin \Sigma \cup \mathcal{X}$. A *context* C is a tree in $T(\Sigma, \mathcal{X} \cup \{z\})$ such that z appears exactly once in the tree. For a context C, we write $C[t]$ instead of $C[z/t]$.

Definition 4: A *context-free tree grammar* (CFTG) is a quadruplet $\mathcal{G} = (N, \Sigma, P, S)$ where

- N is a finite set of non-terminals with rank,
- Σ is a ranked alphabet,
- P is a finite set of production rules in the form $A(x_1, \ldots, x_n) \to t$,
- $S \in N$ is a start non-terminal with rank 0.

A production relation $\Rightarrow_{\mathcal{G}}$ by $\mathcal{G}$ is a binary relation over $T(\Sigma \cup N, \mathcal{X})$ defined by: For $s, s' \in T(\Sigma \cup N, \mathcal{X})$, $s \Rightarrow_{\mathcal{G}} s'$ if and only if there exist a production rule $A(x_1, \ldots, x_n) \to t \in P$ where $\mathrm{rk}(A) = n$, a context $C \in T(\Sigma \cup N, \mathcal{X} \cup \{z\})$, and $t_1, \ldots, t_n \in T(\Sigma \cup N, \mathcal{X})$ such that $s = C[A(t_1, \ldots, t_n)]$, $s' = C[t[x_1/t_1, \ldots, x_n/t_n]]$. Let $\Rightarrow^*_{\mathcal{G}}$ denote the reflexive transitive closure of $\Rightarrow_{\mathcal{G}}$. The tree language $L(\mathcal{G})$ generated by $\mathcal{G}$ is defined as $\{t \in T(\Sigma) \mid S \Rightarrow^*_{\mathcal{G}} t\}$. The binary relation $\rightsquigarrow_{\mathcal{G}}$ is defined as $\{(A, B) \in N \times N \mid A(x_1, \ldots, x_n) \to t \in P$ and B appears in t. $\}$

Definition 5: A *straight-line* CFTG (SLCFTG) is a CFTG $\mathcal{G} = (N, \Sigma, P, S)$ satisfying the followings:

- For any $A \in N$, there exists exactly one production rule $A(x_1, \ldots, x_n) \to t \in P$ and t is linear.
- The relation $\rightsquigarrow_{\mathcal{G}}$ is acyclic.

An SLCFTG is a CFTG that yields only one tree. Let $\mathrm{rhs}(A)$ denote the tree in the right-hand side of the rule with $A \in N$ in its left-hand side. In SLCFTGs, without loss of generality, we assume that in $\mathrm{rhs}(A)$ of each $A \in N$, x_i is the ith variable when traversing in the pre-order. The size $|\mathcal{G}|$ of an SLCFTG $\mathcal{G}$ is defined by $|\mathcal{G}| := \sum_{A \to t \in P} |t|$.

Example 2: The SLCFTG $\mathcal{G} = (\{S, A\}, \{a, b, c, d, e, \#\}, P, S)$ where

$$P = \left\{ \begin{array}{l} S \to a(b(A(e(\#, \#)), c(A(\#), \#)), \#), \\ A(x_1) \to d(\#, e(\#, x_1)) \end{array} \right\}$$

generates the tree (D, λ) in Example 1.

For a non-terminal A and $i \in \{1, \ldots, \mathrm{rk}(A)\}$, let $\mathrm{Vpos}(A, i)$ be the position p such that $A(x_1, \ldots, x_{\mathrm{rk}(A)}) \Rightarrow^* \tilde{t} \in T(\Sigma, \mathcal{X})$ and $\lambda_{\tilde{t}}(p) = x_i$.

Example 3: Consider two rules $A(x_1, x_2) \to a(x_1, B(x_2))$ and $B(x_1) \to b(b(x_1))$ where A and B are non-terminals and a and b are terminals. Then, $\mathrm{Vpos}(A, 1) = 1$ and $\mathrm{Vpos}(A, 2) = 211$ because $A(x_1, x_2) \Rightarrow^* a(x_1, b(b(x_2))) = \tilde{t}$ and $\lambda_{\tilde{t}}(1) = x_1, \lambda_{\tilde{t}}(211) = x_2$.

3. Data Tree Automata

We define deterministic selecting top-down data tree automata with bottom-up look-ahead. We use them as a model of node selection queries for data trees.

Definition 6: A *deterministic selecting top-down data tree automaton with bottom-up look-ahead* over data trees is a tuple $\mathcal{A} = (\Sigma, Q_b, Q_t, q_{b0}, q_{t0}, \Delta_b, \Delta_t, \mathcal{S})$ where

- Σ is a ranked alphabet,
- Q_b and Q_t are finite sets of states,
- $q_{b0} \in Q_b, q_{t0} \in Q_t$ are initial bottom-up and top-down

states,
- Δ_b is a finite set of bottom-up transition rules in the form $f(q_{b1}, q_{b2}) \to (\alpha, \varphi)?q_T : q_F$ where $f \in \Sigma$, $q_T, q_F, q_{b1}, q_{b2} \in Q_b$, $\alpha \in Att$, φ is a predicate on a data value of attribute α,
- Δ_t is a finite set of top-down transition rules in the form $(f, q_b, q_t) \to (q_{t1}, q_{t2})$ where $f \in \Sigma$, $q_t, q_{t1}, q_{t2} \in Q_t$, $q_b \in Q_b$,
- $\mathcal{S} \subseteq \Sigma \times Q_b \times Q_t$, called selection specification, and
- Both Δ_b and Δ_t have no two distinct rules with the same left-hand side.

In this paper, we simply call an automaton defined above a data tree automaton.

Definition 7: Let $\mathcal{A} = (\Sigma, Q_b, Q_t, q_{b0}, q_{t0}, \Delta_b, \Delta_t, S)$ be a data tree automaton. For a data tree $\tau = ((D_t, \lambda_t), \delta)$ over (Σ, Att), an *accepting run* r of $\mathcal{A}$ on τ is a tree $r = (D_r, \lambda_r)$ over $\Sigma \times Q_b \times Q_t$ such that

- $D_r = D_t$, and
- for each node $p \in D_t$,
  - if p is a leaf, there exists a state $q_t \in Q_t$ such that $\lambda_r(p) = (\#, q_{b0}, q_t)$.
  - if p is the root, there exists a state $q_b \in Q_b$ such that $\lambda_r(\epsilon) = (\lambda_t(\epsilon), q_b, q_{t0})$.
  - if p is not a leaf, there exist two transition rules $\lambda_t(p)(q_{b1}, q_{b2}) \to (\alpha, \varphi)?q_T : q_F \in \Delta_b$ and $(\lambda_t(p), q_b, q_t) \to (q_{t1}, q_{t2}) \in \Delta_t$ such that

$$\begin{aligned} & \lambda_r(p) = (\lambda_t(p), q_b, q_t) \\ \wedge\ & (\varphi(\delta(p, \alpha)) \Rightarrow q_b = q_T) \\ \wedge\ & (\neg\varphi(\delta(p, \alpha)) \Rightarrow q_b = q_F) \\ \wedge\ & \forall i \in \{1, 2\}. \exists f. \lambda_r(p \cdot i) = (f, q_{bi}, q_{ti}) \end{aligned}$$

Let $\mathcal{A}_b$ denote the *bottom-up part* $(\Sigma, Q_b, q_{b0}, \Delta_b)$ of $\mathcal{A}$ and an (accepting) *bottom-up run* r_b of $\mathcal{A}_b$ on a data tree $\tau = ((D_t, \lambda_t), \delta)$, is a tree $r_b = (D_{r_b}, \lambda_{r_b})$ over $\Sigma \times Q_b$ satisfying the corresponding conditions for an accepting run of $\mathcal{A}$, forgetting top-down rules Δ_t. Since $\mathcal{A}$ is deterministic, there is at most one accepting run of $\mathcal{A}$ on t. If $r = (D_r, \lambda_r)$ is an accepting run of $\mathcal{A}$ on $\tau = ((D_t, \lambda_t), \delta)$, we say that a node $p \in D_t$ is *selected by* $\mathcal{A}$ if $\lambda_r(p) \in \mathcal{S}$. Let $\mathcal{A}(\tau) = \{p \in D_t \mid \lambda_r(p) \in \mathcal{S}\}$.

For a bottom-up transition rule, if the predicate in the right-hand side is trivially valid or unsatisfiable, we omit the predicate, like $f(q_{b1}, q_{b2}) \to q$ where $q \in Q_b$. In this paper, we restrict for simplicity that a data tree automaton can only test only one attribute of a node in each rule. We can trivially extend the definition to handle a boolean combination of multiple tests on two or more attributes of a node.

Example 4: Let $\Sigma = \{a, b, c, d, e\}$ and $Att = \{x, y, z\}$ Consider $\mathcal{A} = (\Sigma, \{q_1^b, q_2^b\}, \{q_1^t, q_2^t\}, q_1^b, q_1^t, \Delta_b, \Delta_t, \{(d, q_2^b, q_2^t)\})$ where

$$\begin{aligned} &\Delta_b = \\ &\{e(q_1^b, q_1^b) \to (y, \lambda d.d \geq 20)?q_2^b : q_1^b, e(q_1^b, q_2^b) \to q_2^b\} \\ &\cup \{d(q_1^b, q) \to q \mid q \in \{q_1^b, q_2^b\}\} \\ &\cup \{l(q, q_1^b) \to q_1^b \mid l \in \{b, c\}, q \in \{q_1^b, q_2^b\}\} \\ &\cup \{a(q_1^b, q_1^b) \to q_1^b\}, \end{aligned}$$

and

$$\begin{aligned} &\Delta_t = \\ &\quad \{(l, q, q_1^t) \to (q_1^t, q_1^t) \mid l \in \Sigma \setminus \{c\}, q \in \{q_1^b, q_2^b\}\} \\ &\cup \{(c, q, q_1^t) \to (q_2^t, q_1^t) \mid q \in \{q_1^b, q_2^b\}\} \\ &\cup \{(l, q, q_2^t) \to (q_1^t, q_2^t) \mid l \in \{d, e\}, q \in \{q_1^b, q_2^b\}\}. \end{aligned}$$

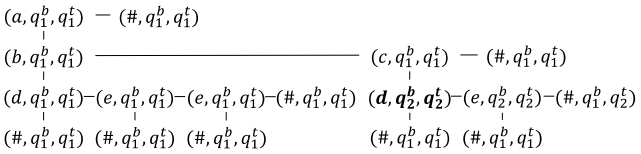


Fig. 2 An accepting run of $\mathcal{A}$ on the data tree in Example 1

Figure 2 shows an accepting run of $\mathcal{A}$ on the data tree $\tau = ((D, \lambda), \delta)$ in Example 1. $\mathcal{A}(\tau) = \{121\}$.

4. Compression

A data tree $\tau = (t, \delta)$ is a pair of the structure t and the valuation function δ. In this paper, we compress only the structure but not the valuation function. We implement a direct update method that traverses the structure part according to a query and refers to the valuation part when a predicate in the query needs a data value. We can apply compression methods for a valuation function such that predicates in queries can be evaluated on the compressed function without decompression. Though we can use the ordinary tree compression with SLCFTGs by handling attribute names and values as terminal symbols, we think that, when compressing a data tree, it is better to compress these two parts separately for the following reasons. Firstly, the number of kinds of tag names are limited even for large XML documents, but data values at different positions are different in many cases. We guess that this lowers compression ratios of compressed data trees using SLCFTGs because the nodes with terminal symbols corresponding data values remain isolated. Secondly, what is the best compression way which can allow direct accesses without decompression depends on what operations are expected to be applied to the compressed data. We think that data values should be compressed depending on their data types or the contexts of names of tags and attributes.

To compress the structure of a data tree, we use an SLCFTG. Note that every SLCFTG generates only one tree (see Sect. 2.2). For a tree t, the compression of t is to construct an SLCFTG $\mathcal{G}$ such that $L(\mathcal{G}) = \{t\}$. Conversely, the decompression of $\mathcal{G}$ is to generate the tree t from $\mathcal{G}$. The compression ratio (of the structure part) is defined by $\frac{|\mathcal{G}|}{|t|}$. It is NP-hard to find the smallest SLCFTG for a given tree. TreeRepair [1] is a tool for compressing trees with no data values. Though it does not guarantee compression to the

smallest SLCFTG, it achieves a good performance on running time and compression ratio.

Definition 8: Let $\tau = (t, \delta)$ be a data tree. $(\mathcal{G}, \delta)$ is a *compressed data tree* of τ where $\mathcal{G}$ is the SLCFTG obtained by compressing t.

5. Direct Update

5.1 Update Instructions

An *update instruction* is a pair $(\mathcal{A}, op)$ where $\mathcal{A}$ is a data tree automaton and op is an operation. For a data tree $\tau = (t, \delta)$ to be updated, op is applied to each position in $\mathcal{A}(\tau)$. An operation op is either of *relabel*, *delete*, *insert-before*, *insert-after*, and *update-value*. Each operation (with arguments) is defined as follows: for a position p selected by $\mathcal{A}$,

- **relabel**(f'), where f' is a symbol, relabels the label of p with f'.
- **delete** replaces the subtree $\lambda_t(p)(t_1, t_2)$ at p with t_2.
- **insert-before**(t'), where $t' = f(t'_1, \#)$ is a tree, replaces the subtree $t|_p$ at p with $f(t'_1, t|_p)$.
- **insert-after**(t'), where $t' = f(t'_1, \#)$ is a tree, replaces the subtree $t|_p = \lambda_t(p)(t_1, t_2)$ at p with $\lambda_t(p)(t_1, f(t'_1, t_2))$.
- **update-value**(α, opr, v'), where α is an attribute name, opr is an arithmetic operator, and v' is a constant natural number, changes the data value v of p on α to $opr(v, v')$.

We have designed the above operations over the binary trees obtained by fcns encoding, corresponding to simplest update operations over unranked trees before the encoding. For a data tree τ and a position p, UpdDT(op, τ, p) denotes the data tree obtained by applying op at p to τ.

For a data tree $\tau = (t, \delta)$ and an update instruction $U = (\mathcal{A}, op)$, we define $U(\tau) :=$ Update$(\tau, \mathcal{A}(\tau), op)$ with

$$\begin{aligned} \text{Update}((t,\delta), \emptyset, op) &= (t, \delta) \\ \text{Update}((t,\delta), \{p\} \uplus P, op) &= \text{Update}((t_1, \delta_2), P_1, op) \end{aligned}$$

where $(t_1, \delta_1) = \text{UpdDT}(op, (t, \delta), p)$, $\delta_2 = \text{UpdPos}(op, \delta_1, p)$, and $P_1 = \text{UpdPos}(op, P, p)$. UpdPos denotes the updates of positions in a valuation function δ or a set P of positions by a given update instruction. The operations delete, insert-before, and insert-after change the structure of the tree, and then some positions with data values may be moved. If such positions exist, we have to update δ. For example, suppose that a position p has a data value v on attribute α. If the position p is moved to p' by an operation, δ is updated to $\delta \setminus \{(p, \alpha, v)\} \cup \{(p', \alpha, v)\}$. UpdPos$(op, \delta, p)$ and UpdPos(op, P, p) are defined as follows.

$$\begin{aligned} &\text{UpdPos}(\textbf{delete}, \delta, p) \\ &= (\delta'' \setminus \{\tilde{p} \mid p \preceq \tilde{p}\} \times Att \times \mathbb{N}) \\ &\qquad \cup \{(p \cdot p'', \alpha, v) \mid (p \cdot 2 \cdot p'', \alpha, v) \in \delta''\}, \\ &\text{UpdPos}(\textbf{delete}, P, p) \\ &= (P \setminus \{\tilde{p} \mid p \preceq \tilde{p}\}) \cup \{p \cdot p'' \mid p \cdot 2 \cdot p'' \in P\} \end{aligned}$$

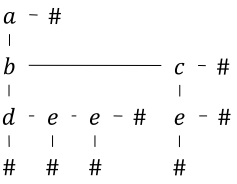


Fig. 3 Structure of $U(\tau)$

Table 2 Valuation function of $U(\tau)$

position	attr	value
11	x	1
112	y	10
112	z	100
1122	y	11
121	y	20

$$\begin{aligned} &\text{UpdPos}(\textbf{insert-before}(t'), \delta, p) \\ &= (\delta \setminus \{\tilde{p} \mid p \preceq \tilde{p}\} \times Att \times \mathbb{N}) \\ &\qquad \cup \{(p \cdot 2 \cdot p'', \alpha, v) \mid (p \cdot p'', \alpha, v) \in \delta\}, \\ &\text{UpdPos}(\textbf{insert-before}(t'), P, p) \\ &= (P \setminus \{\tilde{p} \mid p \preceq \tilde{p}\}) \cup \{p \cdot 2 \cdot p'' \mid p \cdot p'' \in P\} \end{aligned}$$

$$\begin{aligned} &\text{UpdPos}(\textbf{insert-after}(t'), \delta, p) \\ &= (\delta \setminus \{\tilde{p} \mid p \cdot 2 \preceq \tilde{p}\} \times Att \times \mathbb{N}) \\ &\qquad \cup \{(p \cdot 22 \cdot p'', \alpha, v) \mid (p \cdot 2 \cdot p'', \alpha, v) \in \delta\}, \\ &\text{UpdPos}(\textbf{insert-after}(t'), P, p) \\ &= (P \setminus \{\tilde{p} \mid p \cdot 2 \preceq \tilde{p}\}) \cup \{p \cdot 22 \cdot p'' \mid p \cdot 2 \cdot p'' \in P\} \end{aligned}$$

Otherwise UpdPos$(op, \delta, p) = \delta$ and UpdPos$(op, P, p) = P$.

Example 5: Let $U = (\mathcal{A}, \textbf{delete})$ where $\mathcal{A}$ is the data tree automaton in Example 4. Figure 3 and Table 2 show $U(\tau)$ where τ is the data tree in Example 1.

5.2 Proposed Method

Given a compressed data tree $g = (\mathcal{G}, \delta)$ where $\mathcal{G}$ generates t and an update instruction $U = (\mathcal{A}, op)$, we would like to update g to $g' = (\mathcal{G}', \delta')$ such that $U(t, \delta) = (t', \delta')$ and $\mathcal{G}'$ generates t', without generating t from $\mathcal{G}$. Our update procedure consists of two steps: (a) find the positions of τ selected by the data tree automaton $\mathcal{A}$, identifying the corresponding paths in $\mathcal{G}$. (b) apply the update operation op for each selected position in $\mathcal{G}$.

5.2.1 Finding Selected Positions

In this step, we construct the accepting run of a data tree automaton $\mathcal{A}$ over a given compressed data tree $(\mathcal{G}, \delta)$, and remember the paths to the selected positions in $\mathcal{G}$. Basically, this step assigns a bottom-up state and a top-down state for each node in the right-hand side trees of rules of $\mathcal{G}$. If it

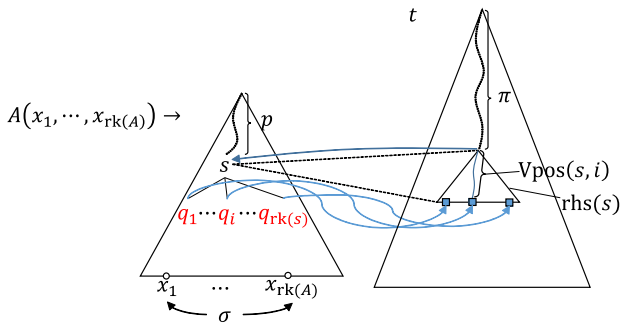


Fig. 4 Execution of BU_EVAL(A, p, π, σ) when s is a non-terminal

reaches a node with a non-terminal symbol A, it moves to the root of rhs(A) and traverses it recursively. Our algorithm consists of two substeps: (1) construct the bottom-up run of $\mathcal{A}_b$, and (2) traverse the bottom-up run, assigning a top-down state to each node in a top-down manner, and remember the selected positions and the paths to the non-terminals A in $\mathcal{G}$ such that there exists a selected position in rhs(A).

(1) Bottom-up traversal

A state assignment is a partial mapping $\sigma : \mathcal{X} \to Q_b$ with a finite domain $\mathrm{dom}(\sigma) \subseteq \mathcal{X}$. We slightly generalize the bottom-up run for a tree in $T(\Sigma, \mathcal{X})$ as follows. For a tree $t = (D_t, \lambda_t) \in T(\Sigma, \mathcal{X}')$, a state assignment σ such that $\mathcal{X}' \subseteq \mathrm{dom}(\sigma)$ and a valuation function δ, the bottom-up run $r = (D_r, \lambda_r)$ on (t, δ) *with* σ is defined as follows: $D_r = D_t$, and for each node $p \in D_t$,

- if $\lambda_t(p) = \#$, $\lambda_r(p) = (\#, q_{b0})$.
- if $\lambda_t(p) = x \in \mathcal{X}'$, $\lambda_r(p) = x$.
- if p is not a leaf, there exist two transition rules $\lambda_t(p)(q_{b1}, q_{b2}) \to (\alpha, \varphi)?q_T : q_F \in \Delta_b$ such that

$$\begin{aligned} &\lambda_r(p) = (\lambda_t(p), q_b, q_t) \\ \wedge\ &(\varphi(\delta(p, \alpha)) \Rightarrow q_b = q_T) \\ \wedge\ &(\neg\varphi(\delta(p, \alpha)) \Rightarrow q_b = q_F) \\ \wedge\ &\forall i \in \{1, 2\}. \\ &(\quad \exists f. \lambda_r(p \cdot i) = (f, q_{bi}) \\ &\quad \vee \exists x \in \mathcal{X}'. (\lambda_t(p \cdot i) = x \wedge \sigma(x) = q_{bi})) \end{aligned}$$

For directly applying the bottom-up part $\mathcal{A}_b = (\Sigma, Q_b, q_{b0}, \Delta_b)$ of a given automaton $\mathcal{A}$ to a given compressed data tree $(\mathcal{G}, \delta)$ of a data tree (t, δ), we design an algorithm BU_EVAL that recursively applies transition rules in Δ_b to the right-hand sides of production rules in $\mathcal{G}$. We begin with BU_EVAL_SIMP (see Algorithm 1), a naive version of BU_EVAL. The algorithm takes a non-terminal A, a position p in rhs(A), a position π in t, and a state assignment σ as arguments and returns the bottom-up run (if exists) r on (t', δ_π) with σ where $\mathrm{rhs}(A)|_p \Rightarrow_{\mathcal{G}}^* t'$ and δ_π is the valuation function derived from δ and π (see its definition between Defs. 2 and 3). Also see Fig. 4. By executing BU_EVAL_SIMP$(S, \epsilon, \epsilon, \bot)$, we can obtain the bottom-up run (if exists) on (t, δ). Note that we need to maintain the position π in t corresponding to the current position in rhs(A) because when we apply a

Algorithm 1 BU_EVAL_SIMP(A, p, π, σ)

Input: Non-terminal A, position p in rhs(A), position π in the whole data tree, and state assignment σ
Output: bottom-up run r on (t', δ_π) with σ where $\mathrm{rhs}(A)|_p \Rightarrow_{\mathcal{G}}^* t'$ and bottom-up state q assigned to the root of r

```
1:  s ← λ_rhs(A)(p)
2:  if s is a variable x then
3:      (r, q) ← (x, σ(x))
4:  else if s is a terminal then
5:      if s = # then
6:          (r, q) ← ((#, q_b0), q_b0) where q_b0 is the initial state
7:      else
8:          (r1, q1) ← BU_EVAL_SIMP(A, p·1, π·1, σ)
9:          (r2, q2) ← BU_EVAL_SIMP(A, p·2, π·2, σ)
10:         if s(q1, q2) → (α, φ)?q_T : q_F ∈ Δ_b then
11:             if φ(δ(π, α)) then
12:                 (r, q) ← ((s, q)(r1, r2), q_T)
13:             else
14:                 (r, q) ← ((s, q)(r1, r2), q_F)
15:             end if
16:         else if s(q1, q2) → q_b ∈ Δ_b then
17:             (r, q) ← ((s, q_b)(r1, r2), q_b)
18:         end if
19:     end if
20: else if s is a non-terminal then
21:     for i = 1 to rk(s) do
22:         (r_i, q_i) ← BU_EVAL_SIMP(A, p·i, π·Vpos(s, i), σ)
23:     end for
24:     σ' ← {x_i ↦ q_i | 1 ≤ i ≤ rk(s)}
25:     (r', q) ← BU_EVAL_SIMP(s, ε, π, σ')
26:     r ← r'[x_i/r_i | 1 ≤ i ≤ rk(s)]
27: end if
28: return (r, q)
```

conditional transition rule $f(q_{b1}, q_{b2}) \to (\alpha, \psi)?q_T : q_F$, we need to know π to refer to the data value $\delta(\pi, \alpha)$.

Clearly, BU_EVAL_SIMP is not more efficient than doing over the data tree before compression. However, many same substructures of the data tree are replaced with a non-terminal by compression. Thanks to this, by constructing the bottom-up run as an SLCFTG instead of a tree, if it visits a non-terminal in the same situation twice or more, we can reuse the previous result for the non-terminal. Here, for a tree t which may contain non-terminals, we define a bottom-up run r on (t, δ) with σ in the same way, adding a case for non-terminals like: if $\lambda_t(p)$ is a non-terminal A with rank n and q_i is assigned to ith child of p for $i \in \{1, \ldots, n\}$, then $\lambda_r(p) = A'$ for a new non-terminal A' and a production $A'(x_1, \ldots, x_n) \to r'$ such that r' is a bottom-up run on $(\mathrm{rhs}(A), \delta_p)$ with $\{x_i \mapsto q_i \mid 1 \le i \le n\}$. We consider that the state assigned to the root of r' is assigned to p in (t, δ).

Assume for simplicity that $\mathcal{A}_b$ does not have any conditional transition rules. Then, for any non-terminal A and any state assignment σ to its arguments, the bottom-up run on (t', δ) with σ where $\mathrm{rhs}(A) \Rightarrow_{\mathcal{G}}^* t'$ and δ is an arbitrary valuation function is uniquely determined independently of δ (if exists). Hence, if we once compute the bottom-up run r on $(\mathrm{rhs}(A), \delta)$ and remember $(A, \sigma)(x_1, \ldots, x_{\mathrm{rk}(A)}) \to r$ as a production rule whose left-hand side is an extended non-terminal (A, σ) with the same rank as A, we can retrieve r when we need to know it afterwards, avoiding the du-

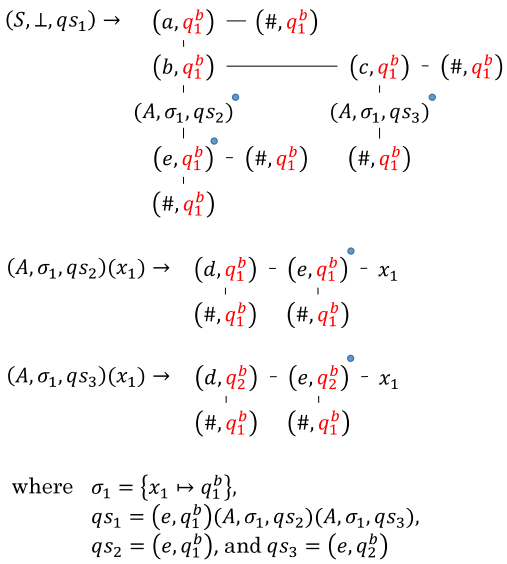
Algorithm 2 BU_EVAL(A, p, π, σ)

Input: Non-terminal A, position p in rhs(A), position π in the whole data tree, and state assignment σ
Output: bottom-up run r on $(\mathrm{rhs}(A)|_p, \delta_\pi)$ with σ, bottom-up state q assigned to the root of r, and sequence qs of symbols of positions in r to which conditional transition rules are applied

1: $s \leftarrow \lambda_{\mathrm{rhs}(A)}(p)$
2: **if** s is a variable x **then**
3: $(r, q, qs) \leftarrow (x, \sigma(x), \epsilon)$
4: **else if** s is a terminal **then**
5: **if** $s = \#$ **then**
6: $(r, q, qs) \leftarrow ((\#, q_{b0}), q_{b0}, \epsilon)$ where q_{b0} is the initial state
7: **else**
8: $(r_1, q_1, qs_1) \leftarrow$ BU_EVAL$(A, p \cdot 1, \pi \cdot 1, \sigma)$
9: $(r_2, q_2, qs_2) \leftarrow$ BU_EVAL$(A, p \cdot 2, \pi \cdot 2, \sigma)$
10: **if** $s(q_1, q_2) \to (\alpha, \varphi)?q_T : q_F \in \Delta_b$ **then**
11: **if** $\varphi(\delta(\pi, \alpha))$ **then**
12: $(r, q, qs) \leftarrow ((s, q)(r_1, r_2), q_T, qs_1 \cdot qs_2 \cdot (s, q))$
13: **else**
14: $(r, q, qs) \leftarrow ((s, q)(r_1, r_2), q_F, qs_1 \cdot qs_2 \cdot (s, q))$
15: **end if**
16: **else if** $s(q_1, q_2) \to q_b \in \Delta_b$ **then**
17: $(r, q, qs) \leftarrow ((s, q_b)(r_1, r_2), q_b, qs_1 \cdot qs_2)$
18: **end if**
19: **end if**
20: **else if** s is a non-terminal **then**
21: $qs \leftarrow \epsilon$
22: **for** $i = 1$ to $\mathrm{rk}(s)$ **do**
23: $(r_i, q_i, qs_i) \leftarrow$ BU_EVAL$(A, p \cdot i, \pi \cdot \mathrm{Vpos}(s, i), \sigma)$
24: $qs \leftarrow qs \cdot qs_i$
25: **end for**
26: $\sigma' \leftarrow \{x_i \mapsto q_i \mid 1 \le i \le \mathrm{rk}(s), (\_, q_i) = \lambda_{r_i}(\epsilon)\}$
27: **if** *brun* has a rule with (s, σ', ϵ) in the lhs **then**
28: Let r' be the rhs of the rule in *brun*
29: $q \leftarrow$ the state appearing as the second element of $\lambda_{r'}(\epsilon)$
30: $qs' \leftarrow \epsilon$
31: **else**
32: $(r', q, qs') \leftarrow$ BU_EVAL$(s, \epsilon, \pi, \sigma')$
33: $brun \leftarrow brun \cup \{(s, \sigma', qs')(x_1, \ldots, x_{\mathrm{rk}(s)}) \to r'\}$
34: **if** $qs' \neq \epsilon$ **then** $qs \leftarrow qs \cdot (s, \sigma', qs')$
35: **end if**
36: $r \leftarrow (s, \sigma', qs')(r_1, \ldots, r_{\mathrm{rk}(s)})$
37: **end if**
38: **return** (r, q, qs)

plicate computation. However, $\mathcal{A}_b$ has conditional transition rules in general and the bottom-up runs of $(\mathrm{rhs}(A), \delta)$ and $(\mathrm{rhs}(A), \delta')$ for different valuation functions δ and δ' may differ because the application of a conditional transition rule may result in different states depending on the data values of δ and δ'. To solve this, we add the sequence qs of $\lambda_r(p)$ for the positions p in rhs(A) to which conditional transition rules are applied (in the post-order) as a component of a new non-terminal. We will call a production rule $(A, \sigma, qs)(x_1, \ldots, x_{\mathrm{rk}(A)}) \to r$ constructed in this way an augmented (production) rule.

The revised algorithm is shown in Algorithm 2. Algorithm 2 uses a global variable *brun* to keep the set of augmented rules constructed during the execution. We first call BU_EVAL$(S, \epsilon, \epsilon, \bot)$ with *brun* empty. If it returns (r, qs), we add $(S, \bot, qs) \to r$ to *brun*. Then, *brun* eventually remembers the SLCFTG, denoted by $\mathcal{G}_{\mathcal{A},\delta}$, where the start symbol is $(S, \bot, qs)$, representing the bottom-up run of $\mathcal{A}$ on t.



where $\sigma_1 = \{x_1 \mapsto q_1^b\}$,
$qs_1 = (e, q_1^b)(A, \sigma_1, qs_2)(A, \sigma_1, qs_3)$,
$qs_2 = (e, q_1^b)$, and $qs_3 = (e, q_2^b)$

Fig. 5 *brun* in Example 6

Algorithm 3 TD_EVAL$(\gamma, p, \pi, q_t, \kappa)$

Input: non-terminal γ of *brun*, position p in rhs(γ), position π in the whole data tree, top-down state q_t, key κ of UDmap

1: $(c, q_b) \leftarrow \lambda_{brun(\gamma)}(p)$
2: **if** c is a variable **then**
3: Append q_t to UDmap(κ).$\mathcal{V}$
4: **else if** c is a terminal but not # **then**
5: **if** $(c, q_b, q_t) \in \mathcal{S}$ **then** ▷ p is selected
6: UDmap(κ).$\mathcal{T} \leftarrow$ UDmap(κ).$\mathcal{T} \cup \{p\}$
7: **end if**
8: **if** $(c, q_b, q_t) \to (q_{t1}, q_{t2}) \in \Delta_t$ **then**
9: TD_EVAL$(\gamma, p \cdot 1, \pi \cdot 1, q_{t1}, \kappa)$
10: TD_EVAL$(\gamma, p \cdot 2, \pi \cdot 2, q_{t2}, \kappa)$
11: **end if**
12: **else if** c is a non-terminal (s, σ, qs) of *brun* **then**
13: $\kappa' \leftarrow (c, q_t)$
14: **if** UDmap(κ') is undefined **then**
15: TD_EVAL$(c, \epsilon, \pi, q_t, \kappa')$
16: **end if**
17: **for** $i = 1$ to $\mathrm{rk}(c)$ **do**
18: $q_{ti} \leftarrow$UDmap(κ').$\mathcal{V}[i]$
19: TD_EVAL$(\gamma, p \cdot i, \pi \cdot \mathrm{Vpos}(s, i), q_{ti}, \kappa)$
20: **end for**
21: **if** UDmap(κ').$\mathcal{T} \neq \emptyset$ or UDmap(κ').$\mathcal{N} \neq \emptyset$ **then**
22: UDmap(κ).$\mathcal{N} \leftarrow$ UDmap(κ).$\mathcal{N} \cup \{(p, \kappa')\}$
23: **end if**
24: **end if**

Example 6: Figure 5 shows *brun* obtained by executing BU_EVAL for the SLCFTG $\mathcal{G}$ given in Example 2 and the bottom-up part of the data automaton $\mathcal{A}$ given in Example 4. *brun* contains the three rules.

(2) Top-down traversal

We design an algorithm TD_EVAL (see Algorithm 3) that recursively applies transition rules in Δ_t to the right-hand sides of production rules in $\mathcal{G}_{\mathcal{A},\delta}$ of the bottom-up run, and remember selected positions. TD_EVAL takes a non-terminal γ of $\mathcal{G}_{\mathcal{A},\delta}$, a position in rhs($\gamma$), a position in the bottom-up

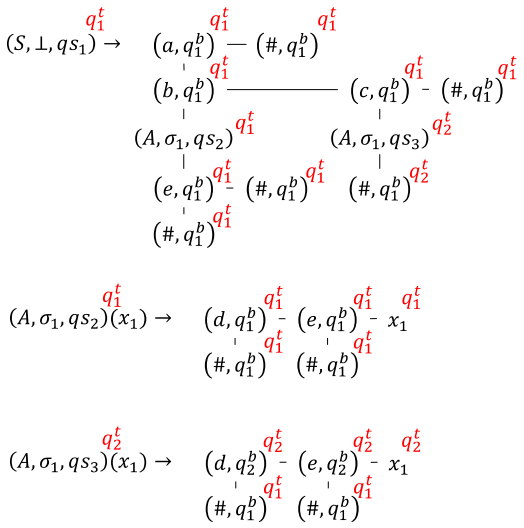


Fig. 6 Top-down state assignment

run, a top-down state and a key of UDmap as arguments. A key of UDmap consists of a non-terminal of $\mathcal{G}_{\mathcal{A},\delta}$ and a top-down state. If the same non-terminal is visited with the same key of UDmap, the accepting run on the right-hand side of the non-terminal is the same. Thus, by remembering the results together with keys, we can avoid duplicate computation. For a key (γ, q_t) where $\gamma = (A, \sigma, qs)$, UDmap remembers a list $\mathcal{V}$ of top-down states assigned to the argument variables of A, a list $\mathcal{T}$ of selected positions in rhs(γ), and a list $\mathcal{N}$ of pairs (p', κ') where p' is the position of a non-terminal γ' such that there is a selected position in rhs(γ') or in a right-hand side reachable from γ' and κ' is a corresponding key. We call TD_EVAL$((S, \bot, qs), \epsilon, \epsilon, q_{t0}, ((S, \bot, qs), q_{t0}))$ with UDmap empty where $(S, \bot, qs)$ is the start symbol of $\mathcal{G}_{\mathcal{A},\delta}$ and q_{t0} is the initial top-down state. After that, we have that $\mathcal{A}(t) = \mathcal{S}(\kappa_I)$ where κ_I is the initial key $((S, \bot, qs), q_{t0})$ and for a key κ, $\mathcal{S}(\kappa)$ is defined as

$$\mathcal{S}(\kappa) = \{p \mid p \in \mathrm{UDmap}(\kappa).\mathcal{T}\} \cup \{p \cdot p' \mid (p, \kappa') \in \mathrm{UDmap}(\kappa).\mathcal{N}, p' \in \mathcal{S}(\kappa')\}.$$

Example 7: Figure 6 shows the assignment of the top-down states when executing TD_EVAL for *brun* given in Example 6 and the data automaton $\mathcal{A}$ given in Example 4. The obtained UDmap is as follows:

key	$\mathcal{V}$	$\mathcal{T}$	$\mathcal{N}$
$((S, \bot, qs_1), q_1^t)$	ϵ	$\emptyset$	$\{(121, ((A, \sigma_1, qs_3), q_2^t))\}$
$((A, \sigma_1, qs_2), q_1^t)$	q_1^t	$\emptyset$	$\emptyset$
$((A, \sigma_1, qs_3), q_2^t)$	q_2^t	$\{\epsilon\}$	$\emptyset$

5.2.2 Application of Update Operation

(1) Updating a grammar

This step is skipped when the update operation is update-value or the set of selected positions is empty, i.e., UDmap has no entries on $\mathcal{T}$ and $\mathcal{N}$ for its initial key.

Algorithm 4 UpdGrammar(κ)

Input: key $\kappa = ((A, \sigma, qs), q_t)$ of UDmap
1: Create a new nonterminal A_c as a copy of A
2: $t_A \leftarrow$ a copy of rhs(A)
3: **for** each $(p, \kappa') \in$ UDmap(κ).$\mathcal{N}$ **do**
4: **if** NT(κ') is undefined **then**
5: UpdGrammar(κ')
6: **end if**
7: $B \leftarrow$ NT(κ')
8: **if** op = **delete then**
9: **for** each $i \in$ RI(B) **do**
10: RI(A_c) $\leftarrow$ RI(A_c) $\cup \{j \mid x_j$ occurs below $p \cdot i$ on $t_A\}$
11: Remove all positions p' s.t. $p \cdot i \preceq p'$ from UDmap(κ).$\mathcal{T}$
12: **end for**
13: Let $t_A|_p = A(t_1, \ldots, t_{\mathrm{rk}(A)})$
14: Let $\{i_1, \ldots, i_{\mathrm{rk}(B)}\} = \{1, \ldots, \mathrm{rk}(A)\} \setminus$ RI(B) such that $i_1 < \cdots < i_{\mathrm{rk}(B)}$
15: Replace the subtree at p on t_A with $B(t_{i_1}, \ldots, t_{i_{\mathrm{rk}(B)}})$
16: **else**
17: Relabel the non-terminal at p on t_A with B
18: **end if**
19: UpdHist(A_c) $\leftarrow$ UpdHist(A_c) $\cup \{(B, p)\}$
20: **end for**
21: **for** each $p \in$ UDmap(κ).$\mathcal{T}$ **do**
22: **if** op = **delete then**
23: RI(A_c) $\leftarrow$ RI(A_c) $\cup \{i \mid x_i$ occurs below p on $t_A\}$
24: **end if**
25: Apply op at p to t_A
26: UpdHist(A_c) $\leftarrow$ UpdHist(A_c) $\cup \{p\}$
27: **end for**
28: **if** there is a copy A' ($\neq A_c$) of A s.t. UpdHist(A') = UpdHist(A_c) **then**
29: NT(κ) $\leftarrow A'$
30: Clear UpdHist(A_c)
31: **else**
32: NT(κ) $\leftarrow A_c$
33: **if** op = **delete then**
34: Renumber the indexes of variables in t_A in the pre-order
35: **end if**
36: $P \leftarrow P \cup \{A_c(x_1, \ldots, x_{\mathrm{rk}(A_c)}) \to t_A\}$
37: **end if**

We can enumerate selected positions by using UDmap. UpdGrammar (see Algorithm 4) takes a key of UDmap, i.e., a non-terminal A of $\mathcal{G}$, a state assignment σ to arguments of A, a state sequence qs, and a top-down state q_t. (Note that (A, σ, qs) is a non-terminal of $\mathcal{G}_{\mathcal{A},\delta}$.) For a given key $\kappa = ((A, \sigma, qs), q_t)$, it creates a copy A_c of A and a copy t_A of rhs(A). A_c is a non-terminal which generates the tree updated according to the key. For each $(p, \kappa') \in$ UDmap(κ).$\mathcal{N}$ where p is a position in rhs(A) and κ' is a key of UDmap, it executes UpdGrammar(κ') and renames the non-terminal at p a new non-terminal. For each $p \in$ UDmap(κ).$\mathcal{T}$ where p is a position in rhs(A), it applies a given update operation at p to t_A. Finally, it checks if the update history UpdHist(A) of t_A coincides with that of another copy of A. If so, it removes A_c and t_A and reuses the existing copy of A instead of A_c because the tree generated from A_c is the same as that from the copy. The map NT remembers the correspondence between the key κ and the corresponding copy of A.

By executing UpdGrammar$((S, \bot, qs), q_{t0})$ with NT and UpdHist empty where $(S, \bot, qs)$ is the start symbol in $\mathcal{G}_{\mathcal{A},\delta}$, we can update the structure of the given data tree.

Lastly, we remove the non-terminals not reachable from the start symbol. Then, we apply a weak variant of the pruning procedure of TreeRepair [1] which removes unprofitable non-terminals as follows. For each non-terminal A such that it is referred only once or rhs(A) contains only a non-terminal and variables, we replace A with rhs(A) and get rid of A and its production rule.

Note that our algorithm does not guarantee the minimality of the updated SLCFTG. In general, the compression ratio gets worse by repeating updates. Our algorithm makes copies of non-terminals and their production rules if needed, and then an update operation is applied. Though some post-processes (removing duplicate non-terminals with the same update history, and pruning unprofitable non-terminals) are conducted to improve the compression ratio, they are locally conducted and thus we cannot find all common parts made after updates in general. We think that the deterioration of compression ratio by updates is unavoidable without detecting the common structures of the whole data tree or keep information on the frequency of occurrences of digrams. The tool grammarRepair [7] can improve the compression ratio of a given SLCFTG. As a way to keep the compression ratio good, we can use the tool when the compression ratio gets much worse by updates.

(2) Updating a valuation function

This step is skipped when the update operation is relabel or the set of selected positions is empty. Otherwise, we execute UpdPos using UDmap for enumerating selected positions.

Example 8: Using UDmap in Example 7 and *brun*, we obtain the SLCFTG with the following rule set P.

$$P = \left\{ \begin{array}{r} S \to a(b(A_1(e(\#,\#)), c(A_2(\#),\#)),\#), \\ A_1(x_1) \to d(\#, e(\#, x_1)) \\ A_2(x_1) \to e(\#, x_1) \end{array} \right\}$$

In the above SLCFTG, each of A_1 and A_2 is referred only once in the first rule. By the pruning procedure, A_1 and A_2 are replaced with rhs(A_1) and rhs(A_2), and then we obtain the SLCFTG with P':

$$P' = \{S \to a(b(d(\#, e(\#, e(\#,\#))), c(e(\#,\#),\#)),\#)\}.$$

Since $\mathcal{A}(t) = \mathcal{S}((S, \bot, qs_1), q_0^t) = \{121\}$, the valuation function is updated to the one shown in Table 2.

5.3 Splitting a Valuation Function

Assume that an XML document has a lot of data values and they are stored in a single file. When an update instruction is processed, loading and restoring unnecessary parts may spend much time. By splitting a valuation function into several files, it can make access to only files containing necessary parts for updating.

We split a valuation function δ into k functions $\delta_1, \ldots, \delta_k$ The position set P_{δ_i} of each δ_i satisfies the following conditions:

Algorithm 5 UpdPos(op, Δ, p)

```
1: δ = find(Δ, p)
2: Let p' such that p = δ.start · p'
3: Replace δ in Δ with UpdPos(op, δ, p')
4: for each δ' ∈ Δ do
5:     if op = delete then
6:         if δ'.start = p ∨ ∃r.δ'.start = p · 1 · r then
7:             Δ ← Δ \ {δ'}
8:         else if ∃r, r'.(δ'.start = p · 2 · r ∧ δ'.end = p · 2 · r') then
9:             (δ'.start, δ'.end) ← (p · r, p · r')
10:        end if
11:    else if op = insert-before then
12:        if ∃r, r'.(δ'.start = p · r ∧ δ'.end = p · r') then
13:            (δ'.start, δ'.end) ← (p · 2 · r, p · 2 · r')
14:        end if
15:    else if op = insert-after then
16:        if ∃r, r'.(δ'.start = p · 2 · r ∧ δ'.end = p · 2 · r') then
17:            (δ'.start, δ'.end) ← (p · 22 · r, p · 22 · r')
18:        end if
19:    end if
20: end for
21: return Δ
```

- there exists a $p_{root} \in P_{\delta_i}$ such that every other position $p' \in P_{\delta_i}$ is a descendant of p_{root}.
- for any $p \in P_{\delta_i}$, every descendant of $p \cdot 1$ is in P_{δ_i}.

We denote p_{root} as $\delta_i.start$ and the last position in the pre-order in P_{δ_i} as $\delta_i.end$. By maintaining the two absolute positions for each split valuation function, we can find which file includes a given position. In each file, each record is positioned relatively to $\delta_i.start$. If an update is applied at an ancestor position of $\delta_i.start$ and a position gap happens, we only have to update $\delta_i.start$ and $\delta_i.end$. Algorithm 5 shows an algorithm UpdPos for split valuation functions where Δ is the set of split valuation functions $\delta_1, \ldots, \delta_k$.

6. Experiments

We implemented a prototype tool for direct update on compressed data trees. We used the XML documents from [22] listed in Table 3. Structures are compressed by TreeRepair [1], and valuation functions of 1-4th documents are divided into about 50 files, and those of 5-7th documents are divided into about 100 files. Table 4 shows update instructions we performed for compressed data trees, where we give XPath expressions equivalent with data tree automata for simplicity. Experiments were performed in the following environment: Intel Core i7-2600 3.40GHz, 16GB RAM, Ubuntu 16.04.3 LTS.

6.1 Updates for Compressed and Uncompressed Data Trees

Tables 5 and 6 show experiment results (CPU time and memory usage (MaxRSS)) of our update method on compressed and uncompressed data trees, respectively. As the tables illustrate, the amount of memory used on compressed data tree is much less than that of used on uncompressed data trees by about 65–91%. This is because the whole tree

Table 3 XML documents and the compressed data tree

File name	File size [kB]	#edges (data tree)	#edges (grammar)	comp. ratio [%]
BaseBall	652	28,305	500	1.77
Shakespeare	7,710	179,689	17,747	9.88
Nasa	25,212	476,645	22,781	4.78
DBLP	134,315	3,332,129	156518	4.70
SwissProt	115,467	2,977,030	247,873	8.33

Table 4 XPath expressions and update operations

#	File name	XPath expression	#selected nodes	operation
1	BaseBall	//PLAYER[HOME_RUNS ≥ 10]	643	delete
2	BaseBall	//LEAGUE// PLAYER[WINS ≥ 15]	30	relabel
3	Shakespeare	//TITLE[text() = ACT I]	37	delete
4	Shakespeare	//TITLE[text() = ACT III]	37	update-value
5	Nasa	//dataset[@subject = astronomy] /altname[1]	2,435	insert-after
6	DBLP	//proceedings[year ≥ 2000]/url	913	insert-before
7	SwissProt	//Entry[@seqlen ≥ 2500]/Species	192	update-value

Table 5 Results on updating compressed data trees

#	parse [ms]	bu [ms]	td [ms]	upd (struct) [ms]	upd (value) [ms]	total [ms]	memory [kB]
1	17	7	0	1	21	46	2,771
2	10	4	0	0	0	14	2,509
3	86	4	4	1	225	320	4,699
4	83	4	4	0	220	311	4,471
5	226	14	7	3	645	895	7,814
6	1,447	1,706	118	222	1,944	5,437	213,494
7	1,627	233	89	0	1,825	3,773	58,937

Table 6 Results on updating uncompressed data trees with decompression and re-compression

#	decompression [ms]	parse [ms]	bu [ms]	td [ms]	upd (struct) [ms]	upd (value) [ms]	compression [ms]	total [ms]	memory [kB]
1	13	34	2	1	2	20	26	59+39	7,829
2	12	31	1	1	5	0	44	38+56	7,726
3	104	188	13	13	44	255	472	513+576	43,626
4	103	191	13	13	0	222	0	439+103	43,626
5	260	538	38	37	142	656	1,132	1,411+1,392	96,285
6	1,698	3,631	426	268	501	1,967	7,464	6,793+9,162	946,575
7	1,547	3,446	246	222	0	1,824	0	5,738+1,547	700,053

Table 7 Comparison of compression ratio after update

	Before update				After update						
					direct update				decomp-update-comp		
#	#edg(dt)	#edg(gr)	ratio[%]	#NT	#edg(dt)	#edg(gr)	ratio[%]	#NT	#edg(gr)	ratio[%]	#NT
1	28,305	500	1.77	38	13,516	481	3.56	45	314	2.32	28
2	28,305	500	1.77	38	28,305	539	1.90	48	546	1.93	37
3	179,689	17,747	9.88	746	179,652	17,769	9.89	751	17,756	9.88	745
4	179,689	17,747	9.88	746	179,689	17,747	9.88	746	17,747	9.88	746
5	476,645	22,781	4.78	1,399	479,080	22,782	4.76	1,399	22,786	4.76	1,403
6	3,332,129	156518	4.70	5,926	3,333,042	156,776	4.70	6,029	156,891	4.71	5,945
7	2,977,030	247,873	8.33	12,024	2,977,030	247,873	8.33	12,024	247,873	8.33	12,024

structures of uncompressed documents are loaded to memory once in the latter case. On the other hand, in the former case, only the compressed tree structure is loaded and the accepting run is compressed with a tree grammar. About the running time, updating the compressed data trees is faster than updating uncompressed ones because the former can avoid duplicate computation.

Table 7 shows compression ratios before and after up-

Table 8 Results on compressed data trees with non-splitting valuation function

#	parse [ms]	bu [ms]	td [ms]	upd (struct) [ms]	upd (value) [ms]	total [ms]	memory [kB]
1	22	7	0	1	37	67	2,774
2	17	3	0	0	0	20	2,508
3	108	4	4	1	318	435	4,462
4	107	4	4	0	302	417	4,435
5	245	14	7	3	741	1,009	7,854
6	1,600	1,699	117	221	4,087	7,723	213,466
7	1,939	231	88	0	5,099	7,358	58,934

Table 9 Results of BaseX

#	time [ms]	memory [kB]
1	534	101,812
2	557	103,620
3	985	140,724
4	1,015	146,540
5	4,004	266,892
6	18,319	1,097,496
7	9,742	882,968

Table 10 Results of our tool

#	time [ms]	memory [kB]
1	46	2,771
2	14	2,509
3	320	4,699
4	311	4,471
5	895	7,814
6	5,437	213,494
7	3,773	58,937

dates. For #4 and #7, the compression ratio does not change because the applied operations are update-value. Except for #1, comparing direct update and decomposition-update-compression, the differences of the compression ratios are small. We think that this is because the size of updated parts are small compared to the whole trees, and few new common parts are made after update. For #2, #5, and #6, the sizes of the output grammars obtained by direct update are a bit smaller than those obtained by decomposition-update-compression. We guess that TreeRepair does not output the smallest grammars in these cases. Lastly, for #1, the size of the data tree reduces to about half by the delete operation. In this case, the compression ratio of the output is 3.56% by direct update, while that by decomposition-update-compression is 2.32%. We guess that due to the large size reduction, the frequency distribution of common parts changes much or a lot of new common parts are made after update. We think that our method can keep the compression ratio good if an update is associated with a relatively small change or does not make a lot of new common parts after update.

6.2 Effects of Splitting on Efficiency

Table 8 shows the result in the case that valuation functions are not split, i.e., all the data values are stored in a single file. Compared with the case that valuation functions are split (Table 5), the case that valuation functions are not split takes more time in parsing and updating valuation function(s).

6.3 Comparison with BaseX

Table 9 shows the total cpu time and memory usage (MaxRSS) for executing the update instructions in Table 4 on BaseX 8.2.3 [9]. GNU time command is used to measure memory usage. The update instructions are given to BaseX by using the XQuery update facility [23]. Table 10 shows only the total time and memory usage on our prototype tool in Table 5. Comparing Tables 9 and 10, our prototype tool executes the update instructions faster with less memory used than BaseX.

7. Conclusion

We have proposed a direct update method for XML documents with data values compressed by SLCFTG. We have implemented a prototype tool and evaluated it by experiments. We confirmed the effectiveness, especially on memory usage, of our update method.

One of future work is to propose a better representation of association between structure and data values. As experimental results show, updating a valuation function spend much time. UpdPos dominates the execution time. If the association is robust against the change of the structure, we can reduce the execution time. Another work is to adopt some compression methods for a valuation function. We think that the best compression method depends on data types and what kind of computation is done for the compressed data values. Hence, we should investigate and organize existing compression methods for data values.

Acknowledgments

The authors thank the anonymous reviewers for their helpful comments on the paper. This work was supported by JSPS KAKENHI Grant Number JP15H02684.

References

[1] M. Lohrey, S. Maneth, and R. Mennicke, "XML tree structure compression using RePair," Information Systems, vol.38, no.8, pp.1150–1167, 2013.

[2] S. Böttcher, R. Hartel, and T. Jacobs, "Fast multi-operations on compressed XML data," Proc. of the 29th British National Conference on Databases (BNCOD 2013), LNCS 7968, pp.149–164, 2013.
[3] T. Goto, T. Onoue, K. Hashimoto, and H. Seki, "Direct update of XML documents compressed by tree grammars (in japanese)," IEICE Technical Report, SS2014-45, 2015.
[4] D.K. Fisher and S. Maneth, "Structural selectivity estimation for XML documents," Proc. of IEEE 23rd International Conference on Data Engineering (ICDE 2007), pp.626–635, 2007.
[5] A. Bätz, S. Böttcher, and R. Hartel, "Updates on grammar-compressed XML data," Proc. of the 28th British National Conference on Databases (BNCOD 2011), pp.154–166, 2011.
[6] S. Böttcher, R. Hartel, T. Jacobs, and M. Jeromin, "ECST - extended context-free straight-line tree grammars," Proc. of the 30th British International Conference on Databases (BICOD 2015), vol.9147, pp.186–198, 2015.
[7] S. Böttcher, R. Hartel, T. Jacobs, and S. Maneth, "Incremental updates on compressed XML," Proc. of the 32nd International Conference on Data Engineering (ICDE 2016), pp.1026–1037, 2016.
[8] K. Hashimoto, S. Nishimura, and H. Seki, "Direct evaluation of selecting tree automata on xml documents compressed with top trees," Proc. of the 4th International Workshop on Trends in Tree Automata and Tree Transducers (TTATT 2016), pp.29–36, 2016.
[9] "BaseX | the xml database." http://basex.org.
[10] P. Buneman, M. Grohe, and C. Koch, "Path queries on compressed XML," Proc. of the 29th International Conference on Very Large Data Bases (VLDB 2003), pp.141–152, 2003.
[11] M. Lohrey, S. Maneth, and E. Noeth, "XML compression via DAGs," Proc. of the 16th International Conference on Database Theory (ICDT 2013), pp.69–80, 2013.
[12] G. Busatto, M. Lohrey, and S. Maneth, "Efficient memory representation of XML document trees," Information Systems, vol.33, no.4-5, pp.456–474, 2008.
[13] S. Böttcher, R. Hartel, and C. Krislin, "Clux - clustering XML subtrees," Proc. of the 12th International Conference on Enterprise Information Systems (ICEIS 2010), pp.142–150, 2010.
[14] A. Jeż, "Faster fully compressed pattern matching by recompression," Proc. of the 39th International Colloquium on Automata, Languages, and Programming (ICALP 2012), pp.533–544, 2012.
[15] A. Jeż and M. Lohrey, "Approximation of smallest linear tree grammar," Proc. of the 31st Symposium on Theoretical Aspects of Computer Science (STACS 2014), pp.445–457, 2014.
[16] M. Lohrey and S. Maneth, "The complexity of tree automata and XPath on grammar-compressed trees," Theoretical Computer Science, vol.363, no.2, pp.196–210, 2006.
[17] P. Bille, I.L. Gørtz, G.M. Landau, and O. Weimann, "Tree compression with top trees," Information and Computation, vol.243, pp.166–177, 2015.
[18] L. Hübschle-Schneider and R. Raman, "Tree compression with top trees revisited," CoRR abs/1506.04499, 2015.
[19] S. Alstrup, J. Holm, K.D. Lichtenberg, and M. Thorup, "Maintaining information in fully dynamic trees with top trees," ACM Transactions on Algorithms (TALG), vol.1, no.2, pp.243–264, 2005.
[20] S. Maneth and F. Peternek, "Compressing graphs by grammars," Proc. of IEEE 32nd International Conference on Data Engineering (ICDE 2016), pp.109–120, 2016.
[21] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi, "Tree automata techniques and applications," 2008. http://tata.gforge.inria.fr/.
[22] "XMLCompBench." http://xmlcompbench.sourceforge.net/Dataset.html.
[23] "W3C XQuery update facility 1.0."

Kenji Hashimoto received the Ph.D. degree in Information and Computer Sciences from Osaka University in 2009. From 2009 to 2013, he was an Assistant Professor of Nara Institute of Science and Technology. Since Oct. 2013, he has been an Assistant Professor in Nagoya University. His research interests include formal language, database theory, and information security.

Ryunosuke Takayama received Bachelor of Engineering and Master of Information Science from Nagoya University in 2015 and 2017. He engaged in compression and direct manipulation of tree structured data during 2015–2017.

Hiroyuki Seki received his Ph.D. degree from Osaka University in 1987. He was an Assistant Professor, and later, an Associate Professor in Osaka University from 1987 to 1994. In 1994, he joined Nara Institute of Science and Technology, where he was a Professor during 1996 to 2013. Currently, he is a Professor in Nagoya University. His current research interests include formal language theory and formal approach to software development.