Validating XML Documents in the Streaming Model with External Memory

CHRISTIAN KONRAD, Univ Paris Diderot, Sorbonne Paris-Cité, LIAFA, CNRS, 75205 Paris, France
FRÉDÉRIC MAGNIEZ, CNRS, LIAFA, Univ Paris Diderot, Sorbonne Paris-Cité, 75205 Paris, France

We study the problem of validating XML documents of size N against general DTDs in the context of streaming algorithms. The starting point of this work is a well-known space lower bound. There are XML documents and DTDs for which p-pass streaming algorithms require $\Omega(N/p)$ space.

We show that when allowing access to external memory, there is a deterministic streaming algorithm that solves this problem with memory space $\mathrm{O}(\log^2 N)$, a constant number of auxiliary read/write streams, and $\mathrm{O}(\log N)$ total number of passes on the XML document and auxiliary streams.

An important intermediate step of this algorithm is the computation of the First-Child-Next-Sibling (FCNS) encoding of the initial XML document in a streaming fashion. We study this problem independently, and we also provide memory efficient streaming algorithms for decoding an XML document given in its FCNS encoding.

Furthermore, validating XML documents encoding binary trees against any DTD in the usual streaming model without external memory can be done with sublinear memory. There is a one-pass algorithm using $\mathrm{O}(\sqrt{N \log N})$ space, and a bidirectional two-pass algorithm using $\mathrm{O}(\log^2 N)$ space which perform this task.

Categories and Subject Descriptors: F.2.2 [**Theory of Computation**]: Analysis of algorithms and problem complexity—*Nonnumerical Algorithms and Problems*

General Terms: Algorithms, Theory

1. INTRODUCTION

Streaming Algorithms. The area of streaming algorithms has experienced tremendous growth over the last decade in many applications. Streaming algorithms sequentially scan the whole input piece by piece in one pass, or in a small number of passes (i.e., they do not have random access to the input), while using sublinear memory space, ideally polylogarithmic in the size of the input. The design of streaming algorithms is motivated by the explosion in the size of the data that algorithms are called upon to process in everyday real-time applications. Examples of such applications occur in bioinformatics for genome decoding, in Web databases for the search of documents, or in network monitoring. The analysis of Internet traffic [Alon et al. 1999], in which traffic logs are queried, was one of the first applications of this kind of algorithm.

There are various extensions of this basic streaming model. One of them gives the streaming algorithm access to an external memory consisting of several read/write streams [Grohe et al. 2005; Grohe and Schweikardt 2005; Grohe et al. 2006]. Then the streaming algorithm is also relaxed and allowed to perform multiple passes in any direction over the input stream and the auxiliary streams. In most of the applications, the number of auxiliary streams is constant and the total number of passes is logarithmic in the input size.

Supported by the French ANR Blanc program under contract ANR-12-BS02-005-001 (RDAM project).

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Databases and XML. Verifying properties or evaluating queries of massive databases is an active and challenging topic. For relational algebra queries against relational databases, the situation is quite clear. There are bidirectional $\mathrm{O}(\log N)$-pass deterministic streaming algorithms with constant memory space and a constant number of auxiliary streams [Grohe et al. 2009] where N is the size of the database. Moreover, the logarithmic number of passes is a necessary condition in order to keep the memory space sublinear, even if randomization is allowed. The latter was initially stated for one-sided error [Grohe et al. 2009] and then extended to two-sided error [Beame et al. 2007; Beame and Huynh-Ngoc 2008].

In the context of data exchange, especially on the Web, Extended Markup Language (XML) is emerging as the standard, and is currently drawing much attention in data management research. Only little is known about XML query processing when only streaming access is allowed to the XML document. For evaluating XQuery and XPath queries against XML documents of size N, only the lower bound has been extended [Grohe et al. 2009; Beame et al. 2007; Beame and Huynh-Ngoc 2008], meaning that $\Omega(\log N)$ passes are necessary. All known upper bounds make one pass over the input and use linear memory in the depth of the document which in the worst case is as large as N.

Validating XML Documents. This paper considers the problem of validating XML documents against a given Document Type Definition (DTD) in a streaming fashion without restrictions on the DTD. An XML document is valid against a DTD if for each node, the sequence of the labels of their children fulfills a regular expression defined in the DTD. Prior works on this topic [Segoufin and Vianu 2002; Segoufin and Sirangelo 2007] essentially try to characterize those DTDs for which validity can be checked by a finite-state automaton, which is a one-pass deterministic streaming algorithm with constant memory. Concerning arbitrary DTDs, two approaches have been considered in [Segoufin and Vianu 2002]. The first one leads to an algorithm with memory space linear in the height of the XML document [Segoufin and Vianu 2002]. The second one consists of constructing a refined DTD of at most quadratic size, which defines a similar family of tree documents as the original one, and against which validation can be done with constant space. Nonetheless, for an existing document and DTD, the latter requires that both documents and DTD, are converted before validation.

A related line of research is the incremental validation of XML documents [Balmin et al. 2004; Barbosa et al. 2004]. Given an XML document and a set of update operations such as insertion and deletion of nodes, the goal is to validate the XML document that is obtained by applying the update operations on the initial document. Balmin et al. [Balmin et al. 2004] show for instance that an XML document with n nodes that is modified by m update operations can be validated against a DTD in time $\mathrm{O}(m \log n)$ and space $\mathrm{O}(n)$ under the assumption that the document was initially valid.

One of the obstacles that prior works had to cope with was the verification of well-formedness of XML documents, meaning that every opening tag matches its same-level closing tag. Due to the work [Magniez et al. 2010], such a verification can be performed now with a constant-pass randomized streaming algorithm with sublinear memory space and no auxiliary streams. In one pass the memory space is $\mathrm{O}(\sqrt{N \log N})$, and collapses to $\mathrm{O}(\log^2 N)$ with an additional pass in reverse direction.

Our Contributions. The starting point of this work is the fact that checking DTD-validity is hard without auxiliary streams. There are DTDs that admit ternary XML documents, and any p-pass bidirectional streaming algorithm which validates those documents against those DTDs requires $\Omega(N/p)$ space. This lower bound even holds if the streaming algorithm makes use of randomness. This lower bound comes from encoding a well-known communication complexity problem, Set-Disjointness, as an

XML validity problem. This lower bound should be well-known, however we are not aware of a complete proof in the literature. In [Grohe et al. 2007], a similar approach using ternary trees with a reduction from Set-Disjointness is used for proving lower bounds for queries. For the sake of completeness we provide a proof in Section 3.1 (**Theorem 3.1**).

On the other hand, it is possible to validate XML documents in one pass and space $\mathrm{O}(d)$, where d is the depth of the XML document (**Theorem A.1**). This algorithm is straightforward and should be well known. Furthermore, we mention that the previously discussed lower bound of $\Omega(N/p)$ can be modified to obtain a lower bound of $\Omega(d/p)$. Therefore, space $\mathrm{O}(d)$ is best possible for one-pass algorithms. For completeness we discuss this algorithm in Appendix A.

We then discuss validity notions, such as extended DTDs (EDTD), which allow the specification of the validity of a node as a function of the grandchildren of that node. Using a further reduction from Set-Disjointness we show that validating XML documents encoding binary trees against those validity schemas requires linear space (**Theorem 3.3**).

For the case of XML documents encoding binary trees, we present in Section 4 three deterministic streaming algorithms for checking validity with sublinear space. As a consequence, the presence of nodes of degree at least 3 is indeed a necessary condition for the linear space lower bound for general documents. We first present two one-pass algorithms with space $\mathrm{O}(\sqrt{N \log N})$ (**Theorem 4.2** and **Theorem 4.4**). The first algorithm, Algorithm 1, processes the input XML document in blocks and is easy to analyze, however, it is not optimal in terms of processing time per letter. The second algorithm, Algorithm 2, uses a stack and has constant processing time per letter. We conjecture that there is a $\Omega(N^{1/2})$ lower bound for one-pass algorithms. With a second pass in reverse direction the memory collapses to $\mathrm{O}(\log^2 N)$ (**Theorem 4.6**). These three algorithms make use of the simple but fundamental fact that in one pass over an XML document each node is seen twice by means of its opening and closing tag. Hence, it is not necessary to remember all opening tags in the stream since there is a second chance to get the same information from their closing tags. Our algorithms exploit this observation. We summarize our streaming algorithms for validating documents that encode binary trees in Figure 1.

Passes	Space	Time	Remark
1	$\mathrm{O}(\sqrt{N \log N})$	$\Omega(\sqrt{N \log N})$	Block Algorithm (Theorem 4.2), simple analysis
1	$\mathrm{O}(\sqrt{N \log N})$	$\mathrm{O}(1)$	Stack Algorithm (Theorem 4.4)
2	$\mathrm{O}(\log^2 N)$	$\mathrm{O}(\log N)$	Bidirectional Algorithm (Theorem 4.6)

Fig. 1. Overview about our streaming algorithms for checking DTD-validity of XML documents that encode binary trees. Time refers to the worst-case processing time between two consecutive read operations from the stream. We did not fully analyze the processing time of the block algorithm, however, since it processes the input in blocks of size $\Theta(\sqrt{N \log N})$, the processing time is $\Omega(\sqrt{N \log N})$.

Then, in Section 5 we present our main result. **Corollary 5.12** states that the validation of any XML document against any DTD can be checked in the streaming model with external memory with poly-logarithmic space, a constant number of auxiliary streams, and $\mathrm{O}(\log N)$ passes over these streams. Validity of a node depends on its children, hence it is crucial to have easy access to the sequence of children of any node. We establish this by computing the First-Child-Next-Sibling (FCNS) encoding, which is an encoding of the XML document as a 2-ranked tree. In this encoding, the sequence of closing tags of the children of a node are consecutive. The computation of this encoding is the hard part of the validation process, and the resource requirements of our validation algorithm stem from this operation (**Theorem 5.6**). Since the FCNS encoding can be seen as a reordering of the tags of the original document, our strategy is to regard this problem as a sorting problem with a particular comparison function. Merge sort can be implemented as a streaming algorithm with auxiliary streams. We use a version that is customized with an adapted merge function. The same idea can be used for FCNS decoding with similar complexity (**Theorem 5.16**). Then, based on the FCNS encoding, verification can be completed either in one pass and $\mathrm{O}(\sqrt{N \log N})$ space (**Theorem 5.11**), or in two bidirectional passes and $\mathrm{O}(\log^2 N)$ space (**Theorem 5.10**). Figure 2 illustrates how our streaming algorithm of Corollary 5.12 for the validation of general XML documents is obtained.

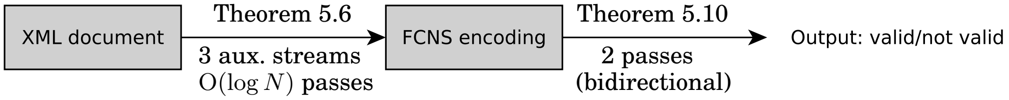


Fig. 2. Schema of our Streaming Algorithm of Corollary 5.12 for checking DTD-validity of arbitrary XML documents. First, the First-Child-Next-Sibling encoding of the input XML document is computed with 3 auxiliary streams and $\mathrm{O}(\log N)$ space. Then, validity of the document is checked with 2 bidirectional passes and $\mathrm{O}(\log^2 N)$ space.

Concerning the computation of the FCNS encoding and the FCNS decoding, we show linear space lower bounds for algorithms that perform one pass over the input and one pass over the output. For decoding, we present an algorithm that uses $\mathrm{O}(\sqrt{N \log N})$ space (**Theorem 5.15**) and performs one pass over the input, but two passes over the output. Furthermore, we show that with 3 auxiliary streams and $\mathrm{O}(\log N)$ passes, both encoding and decoding can be done with $\mathrm{O}(\log N)$ space. For encoding, we conjecture that if no access to auxiliary streams is granted the memory space remains $\Omega(N)$ after any constant number of passes. This would show that decoding is easier than encoding. This suggests a systematic use of the FCNS encoding for large documents, since validity can be checked easily without auxiliary streams and with sublinear space. The applicability of this idea is left as an open question.

LB/UB	Passes	Space	Remark
FCNS encoding:			
Lower Bound	1 pass on input, 1 pass on output	$\Omega(N)$	(Fact 6)
Upper Bound	$\mathrm{O}(\log N)$ passes on 3 auxiliary streams	$\mathrm{O}(\log N)$	(Corollary 5.7)
FCNS decoding:			
Lower Bound	1 pass on input, 1 pass on output	$\Omega(N)$	(Fact 6.2)
Upper Bound	1 pass on input, 2 passes on output	$\mathrm{O}(\sqrt{N \log N})$	(Theorem 5.15)
Upper Bound	$\mathrm{O}(\log N)$ passes on 3 auxiliary streams	$\mathrm{O}(\log N)$	(Theorem 5.16)

Fig. 3. Overview about our results on computing the FCNS encoding and the FCNS decoding. For encoding, an XML document is on the input stream and the goal is to output the FCNS encoding of this document on an output stream. For decoding, the FCNS encoding of an XML document is on the input stream and the goal is to output the original document on an output stream.

Practical relevance. In many applications, XML documents are of bounded depth (for instance a depth of $\mathrm{O}(\log^c N)$ for some constant c is a reasonable assumption), and therefore, the previously mentioned one-pass streaming algorithm with space $\mathrm{O}(d)$, where d is the depth of the XML document, may be sufficient. However, since the depth of an XML document can be as large as $\Theta(N)$, such an algorithm would then require linear space. The space complexity of our streaming algorithm for general documents does not depend on the depth of the underlying XML document. This algorithm is therefore particularly interesting for XML documents with large depth.

Furthermore, note again that our streaming algorithm for general XML documents does not make any assumptions on the structure of the input XML documents and the DTDs. Therefore, this algorithm can be considered to be mainly of theoretical interest, since in practice, the structure of the input XML documents and DTDs may be exploited to obtain algorithms with better space complexity.

Moreover, the authors are not aware of applications that use XML documents that encode binary trees or 2-ranked trees. Our streaming algorithms for the validation of documents that encode binary trees are hence mainly of theoretical interest. Note, however, that we use a modified version of the bidirectional two-pass algorithm for the validation of XML documents that encode binary trees for checking validity of an XML document that is given in its First-Child-Next-Sibling encoding.

Conference version. This work builds on the article [Konrad and Magniez 2012] that was presented at the 15th International Conference on Database Theory 2012 (ICDT 2012). Besides a more detailed presentation of the results of [Konrad and Magniez 2012], this article provides:

- a $\Omega(N/p)$ space lower bound for p-pass streaming algorithms for validating XML documents that encode binary trees against extended DTDs (Theorem 3.3),
- the description of a one-pass streaming algorithm with space $\mathrm{O}(d)$ for validating arbitrary XML documents of depth d against any DTD (Theorem A.1), and
- a linear space lower bound for First-Child-Next-Sibling decoding for streaming algorithms that perform one pass over the input and one pass over the output (Theorem 6.2).

2. PRELIMINARIES

Let Σ be a finite alphabet. The k-th letter of $X \in \Sigma^N$ is denoted by $X[k]$, for $1 \leq k \leq N$, and the consecutive letters of X between positions i and j by $X[i,j]$. A *subsequence* of X is any string $X[i_1]X[i_2]\dots X[i_k]$, where $1 \leq i_1 < i_2 < \dots < i_k \leq N$.

2.1. Streaming model

In streaming algorithms, a *pass* over input $X \in \Sigma^N$ means that X is given as *input stream* $X[1], X[2], \dots, X[N]$, which arrives sequentially, i.e., letter by letter in this order. Streaming algorithms have access to random access memory space, and possibly also to read-write external memory as in [Grohe et al. 2009; Demetrescu et al. 2006]. See also the review in [Grohe et al. 2005]. We assume that any letter of Σ fits into one cell of internal/external memory. The external memory is a collection of auxiliary streams, which are *read/write streams* with sequential access. When needed, we augment the alphabet of auxiliary streams from Σ by k-tuples of elements in $\Sigma \cup [0, 2N]$, for a fixed constant k, which therefore fit into one cell of auxiliary streams.

At the beginning of each pass on a read/write stream, the algorithm decides whether it will perform a read or write pass. The input stream is read-only. On a writing pass, the algorithm can either write a letter, and then move to the next cell, or move directly to the next cell. For the case of bidirectional streaming algorithms, as opposed to unidirectional streaming algorithms where each pass is in the same order, the algorithm can decide the direction of the sequential pass.

For the sake of simplicity, we assume throughout this article that the length of the input is known in advance by the algorithm. Nonetheless, all our algorithms can be adapted to the case in which the length is unknown until the end of a pass. See [Muthukrishnan 2005] for an introduction to streaming algorithms.

Definition 2.1 (*Streaming algorithm*). A $p(N)$-pass *streaming algorithm* $\mathbf{A}$ with $s(N)$ space, $k(N)$ auxiliary streams, $t(N)$ processing time per letter is an algorithm such that for every input stream $X \in \Sigma^N$:

(1) $\mathbf{A}$ has access to $k(N)$ auxiliary read/write streams,
(2) $\mathbf{A}$ performs in total at most $p(N)$ passes on X and auxiliary streams,
(3) $\mathbf{A}$ maintains a memory space of size $s(N)$ letters of Σ and bits while reading X and auxiliary streams,
(4) $\mathbf{A}$ does not exceed a running time of $t(N)$ between two write or read operations.

We say that $\mathbf{A}$ is *bidirectional* if it performs at least one pass in each direction. Otherwise $\mathbf{A}$ is implicitly unidirectional.

If we do not mention the number of auxiliary streams explicitely, then we assume that there none, i.e. $k(N) = 0$. Furthermore, we assume that operations on numbers $N \in [0, 2N]$ can be done in constant time.

Even though all our streaming algorithms are deterministic, we will prove lower bounds that also hold for randomized algorithms. We define a randomized streaming algorithm as follows.

Definition 2.2 (*Randomized streaming algorithm*). A *randomized streaming algorithm* $\mathbf{A}$ with error probability $\epsilon < 1/2$ is a streaming algorithm that has access to an infinite number of independent uniform random bits B, that is $\forall i : \Pr[B[i] = 0] = \Pr[B[i] = 1] = 1/2$. Furthermore, $\mathbf{A}$ is correct with probability at least $1 - \epsilon$ on every input (where the probability is taken over the random bits B).

We say that a randomized streaming algorithm has bounded error if there is an $\epsilon < 1/2$ such that the algorithm is correct on any input with probability at least $1 - \epsilon$.

2.2. XML documents

We consider finite unranked ordered labeled trees t, where each tree node has a label in Σ. From now on, we omit the terms ordered, labeled, and finite. Moreover, the children of every non-leaf node are ordered. k-ranked trees are a special case where each node has at most k children. Binary trees are a special type of 2-ranked tree, where each node is either a leaf or has exactly 2 children. We use the following notations to access the nodes of a tree:

— $\mathrm{root}(t)$: root node of tree t,
— $\mathrm{children}(x)$: (ordered) sequence of children nodes of node x, if x is a leaf then this sequence is empty,
— $\mathrm{fc}(x)$: first child of node x, if x is a leaf then $\mathrm{fc}(x) = \perp$,
— $\mathrm{ns}(x)$: next sibling of node x, if x is a right most (last) child then $\mathrm{ns}(x) = \perp$.

For each label $a \in \Sigma$, we associate its corresponding *opening tag* a and *closing tag* $\overline{a}$, standing for $\langle a \rangle$ and $\langle /a \rangle$ in the usual XML notations. An *XML sequence* is a sequence over the alphabet $\Sigma' = \{a, \overline{a} : a \in \Sigma\}$. The *XML sequence of a tree* t is the sequence of *opening tags* and *closing tags* in the order of a depth first left-to-right traversal of t (Figure 4): when at step i we visit a node with label a top-down (respectively bottom-up), we let $X[i] = a$ (respectively $X[i] = \overline{a}$). Hence X is a word over $\Sigma' = \{a, \overline{a} : a \in \Sigma\}$ of size twice the number of nodes of t. The XML file describing t is unique, and we denote it as $\mathrm{XML}(t)$. We define $\mathrm{XML}(t)$ as a recursive function in Definition 2.3. For a node $x \in t$, we write (ambiguously) x and $\overline{x}$ to denote its opening and closing tag. x is also used to denote its label.

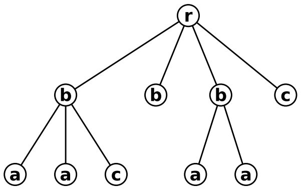


Fig. 4. Let $\Sigma = \{a, b, c\}$, and let t be the tree as above. Then $\mathrm{XML}(t) = rba\overline{a}a\overline{a}c\overline{c}\overline{b}b\overline{b}ba\overline{a}a\overline{a}\overline{b}c\overline{c}\overline{r}$.

Definition 2.3. Let t be an unranked tree, let $x, x_1, \dots, x_n \in t$ be nodes. Then:

$$\begin{aligned}\mathrm{XML}(x) &= x\,\mathrm{XML}(\mathrm{children}(x))\,\overline{x},\\ \mathrm{XML}(x_1, \dots, x_n) &= \mathrm{XML}(x_1)\dots\mathrm{XML}(x_n),\\ \mathrm{XML}(\perp) &= \epsilon,\end{aligned}$$

and we write $\mathrm{XML}(t)$ for $\mathrm{XML}(\mathrm{root}(t))$.

We assume that the input XML sequences X are *well-formed*, namely $X = \mathrm{XML}(t)$, for some tree t. The work [Magniez et al. 2010] legitimates this assumption, since checking well-formedness is at least as easy as any of our algorithms for checking validity. Hence, we could run an algorithm for well-formedness in parallel without increasing the resource requirements. Note that randomness is necessary for checking well-formedness with sublinear space, whereas our algorithms for validation are all deterministic.

Since the length of a well-formed XML sequence is known in advance, we will denote it by $2N$ instead of N. Each opening tag $X[i]$ and matching closing tag $X[j]$ in $X = \mathrm{XML}(t)$ corresponds to a unique tree node v of t. We sometimes denote v either by $X[i]$ or $X[j]$. Then, the *position* of v in X is $\mathrm{pos}(v) = i$. Similarly, $\mathrm{pos}(\overline{v}) = j$.

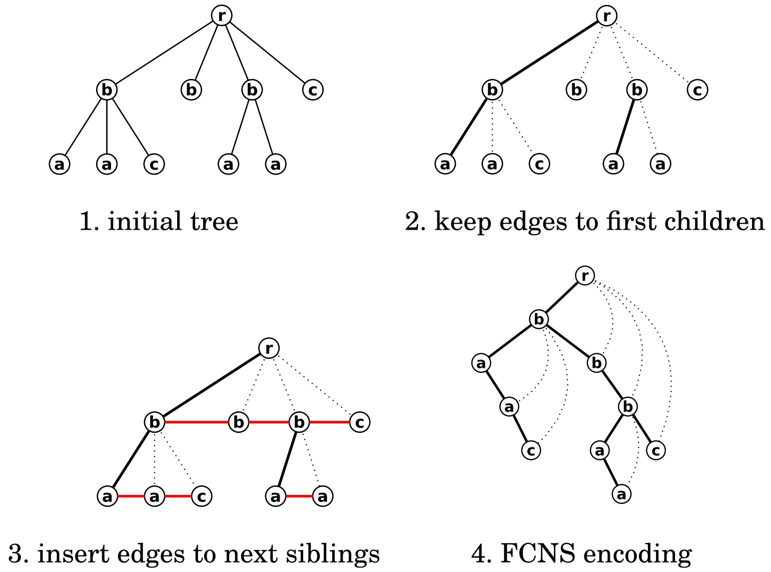
2.3. FCNS encoding and decoding

The *FCNS encoding* (see for instance [Neven 2002]) is an encoding of unranked trees as *extended* 2-ranked trees, where we distinguish left child from right child. This is an extension of ordered 2-ranked trees, since a node may have a left child but no right child, and vice versa. We therefore duplicate the labels $a \in \Sigma$ to a_{L} and a_{R}, in order to denote the *left* and *right* opening/closing tags of a. The FCNS tree is obtained by keeping the same set of tree nodes. The root node of the unranked tree remains the root in the FCNS tree, and we annotate it by default left. The left child of any internal node in the FCNS tree is the first child of this node in the unranked tree if it exists, otherwise it does not have a left child. The right child of a node in the FCNS tree is the next sibling of this node in the unranked tree if it exists, otherwise it does not have a right child. For a tree t, we denote $\mathrm{FCNS}(t)$ the FCNS tree, and $\mathrm{XML}(\mathrm{FCNS}(t))$ the XML sequence of the FCNS encoding of t. Figure 5 illustrates the construction of the FCNS encoding, and we define $\mathrm{XML}(\mathrm{FCNS}(t))$ by means of a recursive function $\mathrm{XML}^{\mathrm{F}}$ in Definition 2.4.

Definition 2.4. Let t be an unranked tree, and let $x \in t$ be a node. Let $D \in \{\mathrm{L}, \mathrm{R}\}$. Then $\mathrm{XML}^{\mathrm{F}}$ is defined as follows:

$$\begin{aligned}\mathrm{XML}^{\mathrm{F}}(x, D) &= x_D\,\mathrm{XML}^{\mathrm{F}}(\mathrm{fc}(x), \mathrm{L})\,\mathrm{XML}^{\mathrm{F}}(\mathrm{ns}(x), \mathrm{R})\,\overline{x_D},\\ \mathrm{XML}^{\mathrm{F}}(\perp, D) &= \epsilon,\end{aligned}$$

and we write $\mathrm{XML}(\mathrm{FCNS}(t))$ for $\mathrm{XML}^{\mathrm{F}}(\mathrm{root}(t), L)$.



1. initial tree

2. keep edges to first children

3. insert edges to next siblings

4. FCNS encoding

Fig. 5. 1: introductory example tree t already shown in Figure 4. 2: removal of all edges except edges to first children. 3: Insertion of edges to next siblings. 4: FCNS encoding of tree t. $\mathrm{XML}^F(t) = r_{\mathrm{L}}b_{\mathrm{L}}a_{\mathrm{L}}a_{\mathrm{R}}c_{\mathrm{R}}\overline{c_{\mathrm{R}}}\,\overline{a_{\mathrm{R}}}\,\overline{a_{\mathrm{L}}}b_{\mathrm{R}}b_{\mathrm{R}}a_{\mathrm{L}}a_{\mathrm{R}}\overline{a_{\mathrm{R}}}\,\overline{a_{\mathrm{L}}}c_{\mathrm{R}}\overline{c_{\mathrm{R}}}\,\overline{b_{\mathrm{R}}}\,\overline{b_{\mathrm{R}}}\,\overline{b_{\mathrm{L}}}\,\overline{r_{\mathrm{L}}}$.

Instead of annotating by left/right, another way to uniquely identify a node as left or right is to insert dummy leaves with a new label $\perp$, and we assume that $\perp \notin \Sigma$. For a tree t, we denote the binary version without annotations and insertion of $\perp$ leaves by $\mathrm{FCNS}^{\perp}(t)$, and the XML sequence of $\mathrm{FCNS}^{\perp}(t)$ by $\mathrm{XML}(\mathrm{FCNS}^{\perp}(t))$. This is illustrated in Figure 6. The two representations can be easily transformed into each other. Depending on the application, we will use the more convenient version.

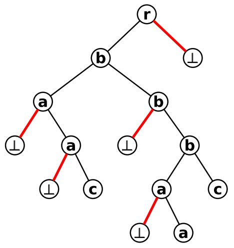


Fig. 6. $\mathrm{FCNS}^{\perp}$ encoding of the example tree. $\mathrm{XML}^{\mathrm{F}\perp}(t) = rba\perp\overline{\perp}a\perp\overline{\perp}c\overline{c}\,\overline{a}\,\overline{a}b\perp\overline{\perp}ba\perp\overline{\perp}a\overline{a}\,\overline{a}c\overline{c}\,\overline{b}\,\overline{b}\,\overline{b}\perp\overline{\perp}\,\overline{r}$.

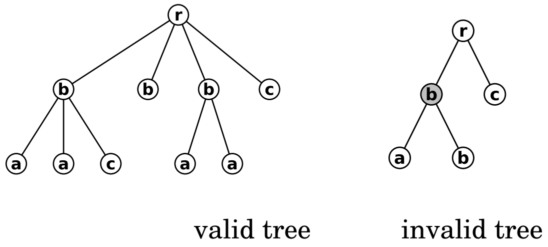
We call the transformation of $\mathrm{XML}(t)$ into $\mathrm{XML}(\mathrm{FCNS}(t))$ ***FCNS encoding***, and the transformation of $\mathrm{XML}(\mathrm{FCNS}(t))$ into $\mathrm{XML}(t)$ ***FCNS decoding***.

2.4. Validity and DTDs

We consider XML validity against some DTD.

Definition 2.5 (*DTD*). A DTD is a triple (Σ, d, s_d) where Σ is a finite alphabet, d is a function that maps Σ-symbols to regular expressions over Σ and $s_d \in \Sigma$ is the start symbol. A tree t satisfies (Σ, d, s_d) if its root is labeled by s_d, and for every node with label a, the sequence $a_1 \ldots a_n$ of labels of its children is in the language defined by $d(a)$.

We illustrate this notion in Figure 7.



$$\Sigma = \{r, a, b, c\},\ s_d = r$$

$$\begin{aligned} d(r) &= b^*c \\ d(a) &= \epsilon \\ d(b) &= a^*c^* \\ d(c) &= \epsilon \end{aligned}$$

Fig. 7. Example of a DTD. The tree on the left is valid agaist the DTD (Σ, d, s_d) shown on the right of this illustration. For each node v of the tree, the sequence of the labels of its children nodes fulfills the regular expression $d(v)$. The tree in the middle is not valid against the DTD on the right: The sequence of the labels of the children of the highlighted node is ab, however, ab does not satisfy the regular expression $d(b) = a^*c^*$.

Throughout the document we assume that DTDs are considerably small and our algorithms have full access to them without accounting this to their space requirements.

Definition 2.6 (VALIDITY). Let D be a DTD. The problem VALIDITY consists of deciding whether an input tree t given by its XML sequence $\mathrm{XML}(t)$ on an input stream is valid against D.

We denote by VALIDITY(2) the problem VALIDITY restricted to input XML sequences describing binary trees.

3. HARDNESS OF SOME NOTIONS OF VALIDITY

In this section, we discuss some lower bounds for checking different notions of validity of XML files. In Section 3.1, we show that there are DTDs that admit ternary trees and checking those requires linear space. In Section 4 we show that checking DTD validity of XML documents encoding binary trees can be done with sublinear space. We conclude that ternary trees are necessary for obtaining a linear space lower bound.

In Section 3.2, we consider validity notions that allow one to express a node's validity as a function of its children and its grandchildren. These validity notions are harder to check than DTD validity in the sense that even checking XML documents encoding binary trees requires linear space.

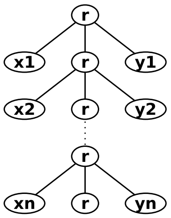
3.1. A linear space lower bound for VALIDITY using ternary trees

We provide now a proof showing that p-pass algorithms require $\Omega(N/p)$ space for checking validity of arbitrary XML files against arbitrary DTDs. Many space lower bound proofs for streaming algorithms are reductions from problems in communication complexity [Alon et al. 1999; Bar-Yossef et al. 2004; Magniez et al. 2010]. For an introduction to communication complexity we refer the reader to [Kushilevitz and Nisan 1997].

Consider a player Alice holding an N bit string $x = x_1 \ldots x_N$, and a player Bob holding an N bit string $y = y_1 \ldots y_N$ both taken from the uniform distribution over $\{0, 1\}^N$. Their common goal is to compute the function $f(x, y) = \bigvee_i x[i] \wedge y[i]$ by exchanging messages. This communication problem is the widely studied problem Set-Disjointness (DISJ).

It is well known that the *randomized communication complexity* with bounded two-sided error of the Set Disjointness function $R(\text{DISJ}) = \Theta(N)$ (see for instance [Kushilevitz and Nisan 1997]). Informally speaking, the randomized communication complexity of a function is the minimal amount of communication in bits that is needed in order to compute the function. In this model, the players Alice and Bob have access to a common string of independent, unbiased coin tosses. The answer is required to be correct with probability at least $2/3$.

We make use of this fact by encoding this problem into an XML validity problem. Consider $\Sigma^{\text{DISJ}} = \{r, 0, 1\}$, the DTD $D^{\text{DISJ}} = (\Sigma^{\text{DISJ}}, d^{\text{DISJ}}, r)$ such that $d^{\text{DISJ}}(r) = 0r0 \,|\, 0r1 \,|\, 1r0 \,|\, \epsilon$, $d^{\text{DISJ}}(0) = \epsilon$, and $d^{\text{DISJ}}(1) = \epsilon$. Given an input x, y as above, we construct an input tree $t(x, y)$ as in Figure 8.



$$d^{\text{DISJ}}(r) = 0r0 \,|\, 0r1 \,|\, 1r0 \,|\, \epsilon$$
$$d^{\text{DISJ}}(0) = d^{\text{DISJ}}(1) = \epsilon$$

Fig. 8. $t(x, y)$ is a hard instance for VALIDITY.

Clearly, $\text{DISJ}(x, y) = 0$ if and only if $\text{XML}(t(x, y))$ is valid with respect to D^{DISJ}.

THEOREM 3.1. *Every p-pass randomized streaming algorithm for* VALIDITY *with bounded error and no auxiliary streams uses $\Omega(N/p)$ space, where N is the input length.*

PROOF. Given an instance $x \in \{0,1\}^N$, $y \in \{0,1\}^N$ of DISJ, we construct an instance for VALIDITY. Then, we show that if there is a p-pass randomized algorithm for VALIDITY using space s with bounded error, then there is a communication protocol for DISJ with the same error and communication $\text{O}(s \cdot p)$. This implies that any p-pass algorithm for VALIDITY requires space $\Omega(N/p)$ since $R(\text{DISJ}) = \Theta(N)$.

Assume that A is a randomized streaming algorithm deciding validity with space s and p passes. Alice generates the first half of $\text{XML}(t(x, y))$, that is $r x_1 \overline{x_1} r x_2 \overline{x_2} \ldots r x_N \overline{x_N} r$ of length $3N + 1$ and executes algorithm A on this sequence using a memory of size $\text{O}(s)$. Alice sends the content of the memory to Bob via message M_A^1. Bob initializes his memory with M_A^1, and continues algorithm A on the second half of $\text{XML}(t(x, y))$, that is $\overline{r} y_N \overline{y_N} \overline{r} \ldots \overline{r} y_2 \overline{y_2} \overline{r} y_1 \overline{y_1} \overline{r}$ of length $3N + 1$. After execution, Bob sends the content of the memory back to Alice via M_B^1. This procedure is repeated at most p times.

This protocol has a total length of $\text{O}(s \cdot p)$ since the size of each message is at most s. Since $R(\text{DISJ}) \in \Theta(N)$, we obtain that $s \cdot p \in \Omega(N)$. The claim follows. □

3.2. Validity notions that allow to relate nodes to their grandchildren

Suppose that a validity schema allows to express a node's validity not only through the labels of its children but also of its grandchildren. Note that this is not the case for DTDs since DTD validity only considers the direct descendants of a node for checking its validity. We show that checking validity against such schemas requires linear space even if the XML document encodes a binary tree, see Theorem 3.3 below.

As in the prior subsection, we encode the communication problem Set-Disjointness into an XML document. Let $x = x_1 \ldots x_N \in \{0,1\}^N$ denote the input of Alice, and let $y = y_1 \ldots y_N \in \{0,1\}^N$ denote the input of Bob. They construct a binary tree $t'(x,y)$ as on the left side of Figure 9.

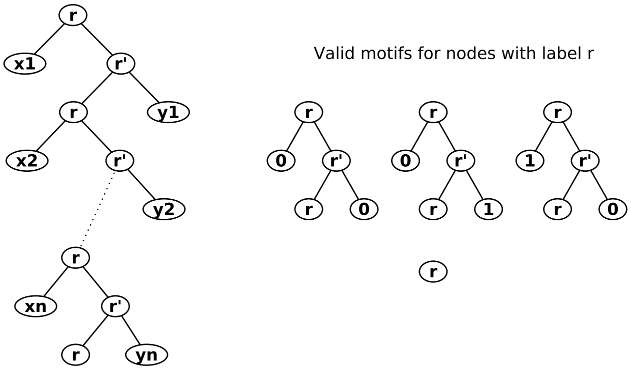


Fig. 9. Left: hard instance $t'(x,y)$ for validity schemas that allow to relate nodes to its grandchildren. Right: the validity constraints for nodes labeled r.

$t'(x,y)$ is valid if the subtrees below a node with label r are as in the right side of Figure 9. Only if this is true then DISJ$(x,y) = 0$. Extended Document Type Definition (EDTD) as well as Relax NG schemas allow to express validity constraints of that kind. XML Schema and DTDs are not powerful enough to express these type of constraints [Martens et al. 2006]. EDTDs were introduced in [Papakonstantinou and Vianu 2000] under the name *specialized DTDs*. They are defined as follows.

Definition 3.2. An extended DTD (EDTD) is a tuple $D = (\Sigma, \Delta, d, s_d, \mu)$, where Δ is a finite set of types, μ is a mapping from Δ to Σ, and (Δ, d, s_d) is a DTD. A tree t satisfies D if $t = \mu(t')$ for some t' satisfying the DTD (Δ, d, s_d).

EDTD validity of the trees $t'(x,y)$ of Figure 9 can be checked by the EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ where

$$\begin{aligned} \Sigma &= \{r, r', 0, 1\}, \Delta = \{r, 0, 1, r'_0, r'_1\}, s_d = r, \\ \mu(r) &= r, \mu(0) = 0, \mu(1) = 1, \mu(r'_0) = r', \mu(r'_1) = r', \text{ and} \\ d(0) &= \epsilon, d(1) = \epsilon, d(r'_0) = r\,0, d(r'_1) = r\,1, d(r) = 0r'_0 | 0r'_1 | 1r'_0 | \epsilon. \end{aligned}$$

THEOREM 3.3. *Every p-pass randomized streaming algorithm validating XML documents encoding binary trees against a validity schema that allows to express a node's validity as a function of its children and its grandchildren with bounded error and no auxiliary streams uses $\Omega(N/p)$ space, where N is the input length.*

PROOF. The proof is identical to the proof of Theorem 3.1, except that the encoding is slightly different. Let $x \in \{0,1\}^N$, $y \in \{0,1\}^N$ be an instance of DISJ. Alice generates the left half of the tree $t'(x,y)$ as follows: $rx_1\overline{x_1}srx_2\overline{x_2}s\ldots rx_n\overline{x_n}sr\overline{r}$. Bob generates the right half of the tree $t'(x,y)$ as follows: $y_n\overline{y_nsr}y_{n-1}\overline{y_{n-1}sr}\ldots y_1\overline{y_1sr}$. A p-pass streaming algorithm with space s checking the two-level validity constraints as on the right side of Figure 9 of $t'(x,y)$ hence solves DISJ with a protocol of length $\mathrm{O}(s \cdot p)$. Since $R(\text{DISJ}) = \Theta(N)$, the result follows. □

4. VALIDITY OF BINARY TREES

For simplicity, we only consider binary trees in this section. A *left opening/closing tag* (respectively *right opening/closing tag*) of an XML sequence X is a tag whose corresponding node is the first child of its parent (respectively second child).

Our algorithms for binary trees can be extended to 2-ranked trees. This requires few changes in the one-pass Algorithms 1 and 2, and the two-pass Algorithm 3 (indeed in the subroutine Algorithm 4) that we do not describe here since they only complicate the presentation and do not affect the essence of the algorithms.

We fix now a DTD $D = (\Sigma, d, s_d)$, and assume that in our algorithms we have access to a procedure check(v, v_1, v_2) that indicates invalidity and aborts if $v_1 v_2$ is not valid against the regular expression $d(v)$. Otherwise it returns without any action. Note that in the case of binary trees, the regular expressions specified in the DTD simplify to disjunctions of allowed pairs and potentially ϵ in case of leaves.

In order to validate an XML document, we ensure validity of all tree nodes. For checking validity of a node v with two children v_1, v_2, we have to relate the labels v_1, v_2 to v. In a *top-down* verification we use the opening tag v of the parent node v for verification, in a *bottom-up* verification we use the closing tag $\overline{v}$ of the parent node v.

4.1. One-pass block algorithm

Algorithms 1 reads the XML document in blocks of size K (we optimize by setting $K = \sqrt{N \log N}$) into memory. Such a block corresponds to a subtree, and the algorithm performs all verifications that are possible within this block. We guarantee that all nodes are verified by ensuring that all substrings $\overline{v_1} v_2$ that correspond to the children of a node v are used for verification. We show in Lemma 4.1 that within a block of any size there is at most one node v with children v_1, v_2 such that $\overline{v_1}$ is in that block but neither the opening tag v nor the closing tag $\overline{v}$ is in that block. Hence, per block all necessary verifications but at most one can be performed. If a pair of tags $\overline{v_1} v_2$ can not be related to their parent node within a block, we store $\overline{v_1} v_2$ and we perform a bottom-up verification upon arrival of the parent node's closing tag $\overline{v}$, see Algorithm 1.

Algorithm 1 Validity of binary trees in 1-pass, block algorithm

```
Require: input stream is a well-formed XML document
 1: $K \leftarrow \sqrt{N \log N}$
 2: $X \leftarrow$ array of size $K+1$, $S \leftarrow$ empty stack
 3: while stream not empty do
 4:   $X \leftarrow$ next $K$ tags on stream
 5:   if $X[K]$ is a closing tag and next tag on stream is an opening tag then
 6:     $X[K+1] \leftarrow$ next tag on stream
 7:   end if
 8:   for all leaves $v$ in $X$ do check($v, \epsilon, \epsilon$) end for
 9:   for all substrings $\overline{v_1}v_2$ of $X$ do {denote the parent node of $v_1, v_2$ by $v$}
10:     if $v \in X$ or $\overline{v} \in X$ then
11:       check($v, v_1, v_2$)
12:     else
13:       push($(v_1, v_2, \mathrm{depth}(v_1))$, $S$)
14:     end if
15:   end for
16:   if stack $S$ not empty then
17:     repeat
18:       $(v_1, v_2, d_1) \leftarrow$ topmost item on stack $S$ {denote the parent node of $v_1, v_2$ by $v$}
19:       if $v \in X$ or $\overline{v} \in X$ then
20:         check($v, v_1, v_2$)
21:         pop $S$
22:       end if
23:     until $v \notin X$ and $\overline{v} \notin X$ or $S$ empty
24:   end if
25: end while
```

In order to compute the depth of tags (as it is required for instance in Line 13), throughout the algorithm we keep track of the current depth with the help of an integer with initial value 0. We increase its value when we encounter an opening tag in the stream and we decrease it when we encounter a closing tag. The depth of a tag is the number of opening tags minus the number of closing tags that precede the tag in the input stream.

The condition in line 10 can be checked as follows. Starting from index i such that $X[i] = \overline{v_1}$, we first traverse X to the left. The first encountered opening tag that has a depth $\mathrm{depth}(v_1) - 1$ (if any) is the opening tag of the parent node v. If the parent node's opening tag is not in X, we then traverse X to the right starting at index i. The first encountered closing tag at level $\mathrm{depth}(v_1) - 1$ (if any) is the closing tag of the parent node v. If X does not comprise any tags at depth $\mathrm{depth}(v_1) - 1$ then the condition evaluates to false. Similarly, the condition in line 19 can be checked. For implementing the condition in line 8 a lookahead of one on the stream might be required if the last tag of X is an opening tag.

LEMMA 4.1. *Let $X[i,j]$ be a block. Then there is at most one left closing tag $\overline{a}$ with parent node p such that:*

$$\mathrm{pos}(p) < i \leq \mathrm{pos}(\overline{a}) \leq j < \mathrm{pos}(\overline{p}). \tag{1}$$

PROOF. For the sake of a contradiction, assume that there are 2 left closing tags $\overline{a}, \overline{b}$ with p being the parent node of a, and q being the parent node of b, for which Inequality 1 holds. Wlog. we assume that $\mathrm{pos}(p) < \mathrm{pos}(q)$. Since $\mathrm{pos}(p) < \mathrm{pos}(q) < \mathrm{pos}(\overline{a})$, q is contained in the subtree of a or $q = a$. This, however, implies that $\mathrm{pos}(\overline{q}) \leq \mathrm{pos}(\overline{a}) < j$ contradicting $\mathrm{pos}(\overline{q}) > j$. □

THEOREM 4.2. *Algorithm 1 is a one-pass streaming algorithm for* VALIDITY(2) *with space* $\mathrm{O}(\sqrt{N \log N})$.

PROOF. To prove correctness, we have to ensure validity of all nodes. Leaves are validated in line 8. Concerning non-leaf nodes, note that all substrings $\overline{v_1}v_2$ are used for validation. Either a node v is validated in line 11 if its opening tag v or its closing tag $\overline{v}$ is in the same block as $\overline{v_1}v_2$, or the node is validated in line 20 if v, $\overline{v_1}v_2$ and $\overline{v}$ are all in different blocks. In this case, the children are pushed on the stack S and the verification is done upon arrival of $\overline{v}$.

Concerning the space, X is of size at most $K+1$. By Lemma 4.1, the stack S grows at most by one element per iteration of the while loop. A stack element requires $\mathrm{O}(\log N)$ storage space since we require to store the depth of the tags which is a number in $[N]$. Since there are $\mathrm{O}(N/K)$ iterations, the total memory requirements are $\mathrm{O}(K + N/K \log(N))$ which is minimized for $K = \sqrt{N \log N}$. □

4.2. One-pass algorithm using a stack

We now present a second one-pass streaming algorithm, Algorithm 2, for checking validity of XML documents that encode binary trees. This algorithm has the same space complexity as the block algorithm, Algorithm 1, of the previous section, however, it has optimal (constant) processing time per letter.

Algorithm 2 Validity of binary trees in 1-pass, stack algorithm

```
Require: input stream is a well-formed XML document
 1: $d \leftarrow 0, S \leftarrow$ empty stack
 2: $K \leftarrow \sqrt{N \log N}$
 3: while stream not empty do
 4:   $x \leftarrow$ next tag on stream
 5:   if $x$ is an opening tag $c$ then
 6:     if $x$ is a leaf then check$(c, \epsilon, \epsilon)$ end if
 7:     if $S$ has on top $(a, -1), (\overline{b}, d)$ then
 8:       check$(a, b, c)$; pop $S$ {Top-down verification}
 9:     end if
10:     if $|\{(a, -1) \in S \mid a$ opening $\}| \geq K$ then
11:       remove bottom-most $(a, -1)$ in $S$, where $a$ is an opening tag
12:     end if
13:     $d \leftarrow d + 1$
14:     push $((x, -1), S)$
15:   else if $x$ is a closing tag $\overline{c}$ then
16:     $d \leftarrow d - 1$
17:     if S has on top $(\overline{a}, d + 1), (\overline{b}, d + 1)$ then
18:       check $(c, a, b)$ {Bottom-up verification}
19:       pop $S$, pop $S$
20:     else if $S$ has on top $(\overline{b}, d + 1)$ then
21:     pop $S$
22:     end if
23:     if $S$ has on top $(c, -1)$ then pop $S$ end if
24:     push $((x, d), S)$
25:   end if
26: end while
```

Algorithm 2 performs top-down and bottom-up verifications. It uses a stack onto which it pushes all opening tags in order to perform top-down verifications once the information of the children nodes arrives on the stream. $\overline{v_1}v_2$ forms a substring of the input, hence top-down verification requires only the storage of the opening tag v since the labels of the children arrive in a block. The algorithm's space requirement depends on a parameter K (we optimize by setting $K = \sqrt{N \log N}$). Once the number of opening tags on the stack is about to exceed K, we remove the bottom-most opening tag. The corresponding node will then be verified bottom-up. Note that $\overline{v_2 v}$ forms a substring of the input. Hence, for bottom-up verifications it is enough to store the label of the left child v_1 on the stack since the label of the right child arrives in form of a closing tag right before the closing tag of the parent node. See Algorithm 2 for details.

For the unique identification of closing tags on the stack, we have to store them with their depth in the tree. A stack item corresponding to a closing tag requires hence $\mathrm{O}(\log N)$ space. Opening tags don't require the storage of their depth (we store the default depth -1).

The query in line 6 can be implemented by a lookahead of 1 on the stream. The opening tag x corresponds to a leaf only if the subsequent tag in the stream is the corresponding closing tag $\overline{x}$.

Figure 10 visualizes the different cases with their stack modifications appearing in Algorithm 2.

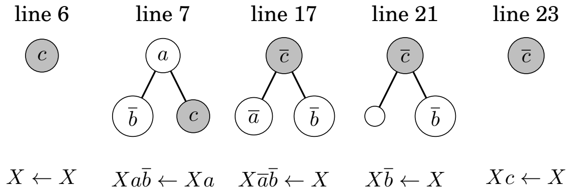


Fig. 10. Visualization of the different conditions in Algorithm 2 with the applied stack modifications. X represents the bottom part of the stack. Note that Algorithm 2 pushes the currently treated tag c or $\overline{c}$ on the stack in Line 14 or Line 24. c or $\overline{c}$ corresponds to the highlighted node.

Fact 1 (which can be easily proved by induction) and Lemma 4.3 concern the structure of the stack S used in Algorithm 2.

FACT 1. *Let* $S = (x_1, d_1), \dots (x_k, d_k)$ *be the stack at the beginning of the while loop in line 3. Then:*

(*1*) $\mathrm{pos}(x_1) < \mathrm{pos}(x_2) \cdots < \mathrm{pos}(x_k)$*,*
(*2*) $\mathrm{depth}(x_1) \leq \mathrm{depth}(x_2) \cdots \leq \mathrm{depth}(x_k) \leq d$*. Moreover, if* $\mathrm{depth}(x_i) = \mathrm{depth}(x_{i+1})$ *then* x_i *is the left sibling of* x_{i+1}*,*
(*3*) *The sequence* $x_1 \dots x_k$ *satisfies the regular expression* $\overline{a}^* b^* (\epsilon \,|\, \overline{c} \,|\, \overline{d}\overline{e})$*, where* $\overline{a}^*$ *are left closing tags,* b^* *are opening tags,* $\overline{c}$ *is a closing tag,* $\overline{d}$ *is a left closing tag, and* $\overline{e}$ *is a right closing tag.*
(*4*) *A left closing tag* $\overline{a}$ *is removed from* S *just after its parent node is verified.*

LEMMA 4.3. *Let* $S = (x_1, d_1), \dots (x_k, d_k)$ *be the stack at the beginning of the while loop in line 3. Let* $(\overline{c_i}, d_i), (\overline{c_{i+1}}, d_{i+1})$ *be two consecutive left closing tags in* S *such that* $(\overline{c_{i+1}}, d_{i+1})$ *is not the topmost left closing tag. Then* $\mathrm{pos}(\overline{c_{i+1}}) \geq \mathrm{pos}(\overline{c_i}) + 2K$.

PROOF. Denote by $X = X[1]X[2] \dots X[2N]$ the input stream. Since $\overline{c_{i+1}}$ is not the topmost left closing tag in S, the algorithm has already processed the right sibling opening tag $X[\mathrm{pos}(\overline{c_{i+1}}) + 1]$ of $\overline{c_{i+1}}$. By Item 4 of Fact 1, no verification has been done of the parent of $\overline{c_{i+1}}$, since $\overline{c_{i+1}}$ is still in S. Therefore, the parent's opening tag $X[k]$ of $\overline{c_{i+1}}$ has been deleted from S, where $\mathrm{pos}(\overline{c_i}) < k < \mathrm{pos}(\overline{c_{i+1}})$. This can only happen if at least K opening tags have been pushed on S between $X[k]$ and $\overline{c_{i+1}}$. Since these

K opening tags must have been closed between $X[k]$ and $\overline{c_{i+1}}$ we obtain $\mathrm{pos}(\overline{c_{i+1}}) \geq \mathrm{pos}(\overline{c_i}) + 2K$. $\square$

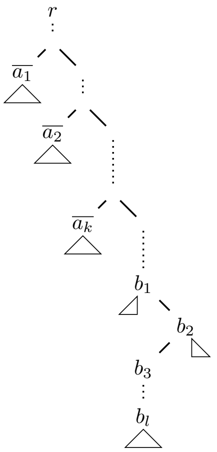


Fig. 11. Visualization of the structure of the stack used in Algorithm 2. The stack fulfills the regular expression $\overline{a}^*b^*(\epsilon \mid \overline{c} \mid \overline{d}\overline{e})$, compare Item 3 of Fact 1. The $(\overline{a_i})_{i=1\ldots k}$ are closing tags whose parents' nodes were not verified top-down. For $j > i$, a_j is connected to a_i by the right sibling of a_i. The $(b_i)_{i=1\ldots l}$ form a sequence of opening tags such that b_i is the parent node of b_{i+1}. On top of the stack might be one or two closing tags depending on the current state of the verification process.

Fact 1 and Lemma 4.3 provide more insight in the stack structure and are used in the proof of Theorem 4.4. Item 3 of Fact 1 states that the stack basically consists of a sequence of left closing tags which are the left children that are needed for bottom-up verifications of nodes that could not be verified top-down. This sequence is followed by a sequence of opening tags for which we still aim a top-down verification. The proof of Lemma 4.3 explains the fact that the two sequences are strictly separated: a left-closing tag $\overline{v_1}$ only remains on the stack if at the moment of insertion there are no opening tags on the stack.

THEOREM 4.4. *Algorithm 2 is a one-pass streaming algorithm for* VALIDITY(2) *with space* $\mathrm{O}(\sqrt{N \log N})$ *and* $\mathrm{O}(1)$ *processing time per letter.*

PROOF. To prove correctness, we have to ensure validity of all nodes. Each leaf is correctly validated upon arrival of its opening tag in line 6. Concerning non-leaf nodes, firstly, note that all closing tags are pushed on S in line 24, in particular all closing tags of left children appear on the stack. The algorithm removes left closing tags only after validation of its parent node, no matter whether the verification was done top-down or bottom-up, compare Item 4 of Fact 1. Emptiness of the stack after the execution of the algorithm follows from Item 2 of Fact 1 and implies hence the validation of all non-leaf nodes.

For the space bound, Line 10 guarantees that the number of opening tags in S is always at most K. We bound the number of closing tags on the stack by $\frac{N}{K} + 2$. Item 3 of Fact 4.3 states that the stack contains at most one right closing tag. From Item 4 of Fact 4.3 we deduce that S comprises at most $\frac{N}{K} + 1$ left closing tags, since the stream is of length $2N$, and the distance in the stream of two consecutive left closing tags that reside on S except the top-most one is at least $2K$. A closing tag with depth $(a, d) \in \Sigma' \times [N]$ requires $\mathrm{O}(\log N)$ space, an opening tag requires only constant space. Hence the total space requirements are $\mathrm{O}((\frac{N}{K}+2)\log N + K)$ which is minimized for $K = \sqrt{N \log N}$.

Concerning the processing time per letter, the algorithm only performs a constant number of local stack operations in one iteration of the while loop. □

Remark Algorithm 2 can be turned into an algorithm with space complexity $\mathrm{O}(\sqrt{D \log D})$, where D is the depth of the XML document. If D is known beforehand, it is enough to set $K = \sqrt{D \log D}$ in line 2. If D is not known in advance, we make use of an auxiliary variable D' storing a guess for the document depth. Initially we set $D' = C$, $C > 0$ some constant, we set $K = \sqrt{D' \log D'}$, and we run Algorithm 2. Each time d exceeds D', we double D', and we update K accordingly.

This guarantees that the number of opening tags on the stack is limited by $\mathrm{O}(\sqrt{D \log D})$. Since we started with a too small guess for the document depth, we may have removed opening tags that would have remained on the stack if we had chosen the depth correctly. This leads to further bottom-up verifications, but no more than $\mathrm{O}(\sqrt{D / \log D})$ guaranteeing $\mathrm{O}(\sqrt{D \log D})$ space.

4.3. Two-pass algorithm

Algorithm 3 Two-pass algorithm validating binary trees

```
run **Algorithm 4** reading the stream from left to right
run **Algorithm 4** reading the stream from right to left, where opening tags are interpreted as
closing tags, and vice versa.
```

Algorithm 4 Validating nodes with size(left subtree) $\geq$ size(right subtree)

```
$l \leftarrow 0$; $n \leftarrow 0$; $S \leftarrow$ empty stack
while stream not empty do
    $x \leftarrow$ next tag on stream (and move stream to next tag)
    $y \leftarrow$ next tag on stream, without consuming it yet
    $n \leftarrow n + 1$
    if $x$ is an opening tag $c$ then
        $l \leftarrow l + 1$
        if $y = \overline{c}$ then check($c, \epsilon, \epsilon$) end if
    else {$x$ is a closing tag $\overline{c}$}
        $l \leftarrow l - 1$
        if $S$ has on top $(\cdot, \cdot, l + 1, \cdot)$ then
            $(\overline{a}, b, \cdot, \cdot) \leftarrow$ pop from $S$; check($c, a, b$)
        end if
        if $y$ is an opening tag $d$ then
            push ($\overline{c}$, $d$, $l$, $n$) to $S$
        end if
    end if
    while there is $s_1 = (\cdot, \cdot, \cdot, n_1)$ just below $s_2 = (\cdot, \cdot, \cdot, n_2)$ in $S$ with $n - n_2 > n_2 - n_1$ do
        delete $s_2$ from $S$
    end while
end while
```

The bidirectional two-pass algorithm, Algorithm 3, uses a subroutine that checks in one-pass validity of all nodes whose left subtree is at least as large as its right subtree. Feeding into this subroutine the XML document read in reverse direction and interpreting opening tags as closing tags and vice versa, it checks validity of all nodes whose right subtree is at least as large as its left subtree. In this way all tree nodes get verified.

The subroutine performs only checks in a bottom-up fashion, that is, the verification of a node v with children c_1, c_2 makes use of the tags $\overline{c_1}$ and c_2 (which are adjacent in the XML document and hence easy to recognize) and the closing tag of $\overline{v}$. When $\overline{c_1}, c_2$ appear in the stream, a 4-tuple consisting of $\overline{c_1}, c_2, \mathrm{depth}(c_1)$ and $\mathrm{pos}(\overline{c_1})$ is pushed on the stack. Upon arrival of $\overline{v}$, $\mathrm{depth}(c_1)$ is needed to identify c_1, c_2 as the children of v. $\mathrm{pos}(\overline{c_1})$ is needed for cleaning the stack: with the help of the pos values of the stack items, we identify stack items whose parents' nodes have larger right subtrees than left subtrees, and these stack items get removed from the stack. In so doing, we guarantee that the stack size does not exceed $\log(N)$ elements which is an exponential improvement over the one-pass algorithm.

Note that the reverse pass can be done independently of the first one, for instance in parallel to the first pass.

Figure 12 visualizes the different cases in Algorithm 4.

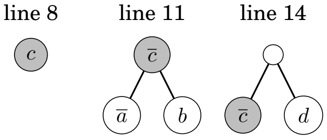


Fig. 12. Visualization of the different conditions in Algorithm 4. The incoming tag x corresponds to the highlighted node.

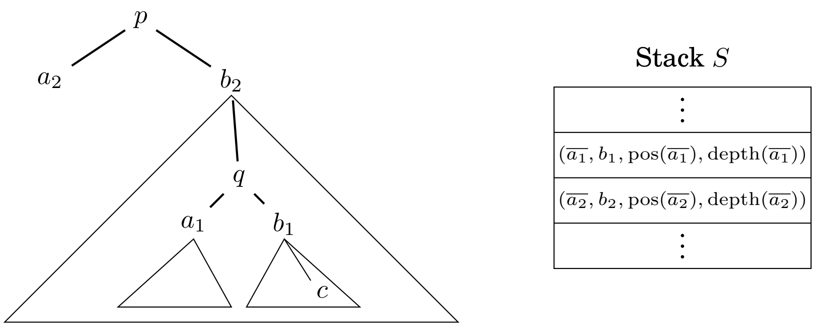
We highlight some properties concerning the stack used in Algorithm 4.

FACT 2. *S in Algorithm 4 satisfies the following:*

(*1*) *If* $(\overline{a_2}, b_2, \mathrm{depth}(\overline{a_2}), \mathrm{pos}(a_2))$ *is below* $(\overline{a_1}, b_1, \mathrm{depth}(\overline{a_1}), \mathrm{pos}(a_1))$ *in* S*, then* $\mathrm{pos}(\overline{a_2}) < \mathrm{pos}(\overline{a_1})$*,* $\mathrm{depth}(\overline{a_2}) < \mathrm{depth}(\overline{a_1})$*, and* a_1, b_1 *are in the subtree of* b_2*.*

(*2*) *Consider* l *at the end of the while loop in line 20. Then there are no stack elements* $(\cdot, \cdot, l', \cdot)$ *with* $l' > l$*.*

Figure 13 illustrates the relationship between two consecutive stack elements as discussed in Item 1 of Fact 2.



$(\overline{a_1}, b_1, \mathrm{pos}(\overline{a_1}), \mathrm{depth}(\overline{a_1}))$

$(\overline{a_2}, b_2, \mathrm{pos}(\overline{a_2}), \mathrm{depth}(\overline{a_2}))$

Fig. 13. c is the current element under consideration in Algorithm 4. a_1, b_1 is in the subtree of b_2, compare Item 1 of Fact 2.

LEMMA 4.5. *Algorithm 4 verifies all nodes q whose left subtree is at least as large as its right subtree.*

PROOF. Let q be such a node. Let a_1, b_1 be the children of q. Then it holds that

$$\mathrm{pos}(\overline{a_1}) - \mathrm{pos}(a_1) \geq \mathrm{pos}(\overline{b_1}) - \mathrm{pos}(b_1), \tag{2}$$

since the size of the left subtree of q is at least as large as the size of the right subtree.

Upon arrival of $\overline{a_1}$ Algorithm 4 pushes the 4-tuple $t = (\overline{a_1}, b_1, \mathrm{pos}(\overline{a_1}), \mathrm{depth}(a_1))$ onto the stack S. We have to show that t remains on the stack until the arrival of $\overline{q}$. More precisely, we have to show that the condition in line 18 is never satisfied for $s_2 = t$. Since the algorithm never deletes the bottom-most stack item, we consider the case where there is a stack item $(\overline{a_2}, b_2, \mathrm{pos}(\overline{a_2}), \mathrm{depth}(a_2))$ just below t. Item 1 of Fact 2 tells us that a_1, b_1 are in the subtree of b_2. Let x be the current tag under consideration such that $\mathrm{pos}(b_1) < \mathrm{pos}(x) < \mathrm{pos}(\overline{q})$. The situation is visualized in Figure 13.

According to the condition of line 18, t gets removed from the stack if

$$\mathrm{pos}(x) - \mathrm{pos}(\overline{a_1}) > \mathrm{pos}(\overline{a_1}) - \mathrm{pos}(\overline{a_2}). \tag{3}$$

Note that the left side of Inequality 3 is a lower bound on the size of the right subtree of q. Furthermore, the right side of Inequality 3 is an upper bound for the size of the left subtree of q.

Using $\mathrm{pos}(x) - \mathrm{pos}(\overline{a_1}) \leq \mathrm{pos}(\overline{b_1}) - \mathrm{pos}(b_1) + 1$ and $\mathrm{pos}(\overline{a_1}) - \mathrm{pos}(\overline{a_2}) > \mathrm{pos}(\overline{a_1}) - \mathrm{pos}(a_1)$, Inequality 3 contradicts Inequality 2 which shows that t remains on the stack until the arrival of $\overline{q}$. Item 2 of Fact 2 guarantees that there is no other stack element on top of t upon arrival of $\overline{q}$. This guarantees the verification of node q and proves the lemma. □

THEOREM 4.6. *Algorithm 3 is a bidirectional two-pass streaming algorithm for* VALIDITY(2) *with space* $\mathrm{O}(\log^2 N)$ *and* $\mathrm{O}(\log N)$ *processing time per letter.*

PROOF. To prove correctness of Algorithm 3, we ensure that all nodes get verified. By Lemma 4.5, in the first pass, all nodes with a left subtree being at least as large as its right subtree get verified. The second pass ensures then verification of nodes with a right subtree that is at least as large as its left subtree.

Next, we prove by contradiction that for any current value of variable n in Algorithm 4, the stack contains at most $\log(n)$ elements. Assume that there is a stack configuration of size $t \geq \log(n) + 1$. Let $(n_1, n_2 \ldots, n_t)$ be the sequence of the fourth parameters of the stack elements. Since these elements are not yet removed, due to line 18 of Algorithm 4, it holds that $n - n_i \leq n_i - n_{i-1}$, or equivalently $n_i \geq 1/2(n + n_{i-1})$, for all $1 < i \leq t$. Since $n_1 \geq 1$, we obtain that $n_i \geq \frac{2^i - 1}{2^i} n + \frac{1}{2^i}$, and, in particular, $n_{t-1} \geq (n-1) + \frac{1}{n}$. Since all n_i are integers, it holds that $n_{t-1} \geq n$. Furthermore, since $n_t > n_{t-1}$, we obtain $n_{\log n + 1} \geq n + 1$ which is a contradiction, since the element at position $n + 1$ has not yet been seen.

Since $n \leq 2N$ and the size of a stack element is in $\mathrm{O}(\log n)$, Algorithm 4 uses space $\mathrm{O}(\log^2 N)$. This also implies that the while-loop at line 18 of Algorithm 4 can only be iterated $\mathrm{O}(\log n)$ times during the processing of a tag on the stream. The processing time per letter is then $\mathrm{O}(\log N)$, since we assume that operations on the stack run in constant time. □

5. VALIDITY OF GENERAL TREES

In the following, we provide streaming algorithms for the FCNS encoding which is the transformation of $\mathrm{XML}(t)$ to $\mathrm{XML}(\mathrm{FCNS}(t))$, and for the FCNS decoding which is the transformation of $\mathrm{XML}(\mathrm{FCNS}(t))$ to $\mathrm{XML}(t)$, see the definition in Section 2.3.

5.1. FCNS encoding

In this section, we are interested in computing the transformation $\mathrm{XML}(t) \to \mathrm{XML}(\mathrm{FCNS}(t))$. Our strategy is to compute the subsequence of opening tags of

$\mathrm{XML}(\mathrm{FCNS}(t))$ (discussed in Subsection 5.1.1) and the subsequence of closing tags (discussed in Subsection 5.1.2) of $\mathrm{XML}(\mathrm{FCNS}(t))$ independently, and merge them afterwards (discussed in Subsection 5.1.3).

5.1.1. Computing the sequence of opening tags.

First, we provide a lemma that shows that the sequence of opening tags in $\mathrm{XML}(t)$ and $\mathrm{XML}(\mathrm{FCNS}(t))$ coincide. The proof of Lemma 5.1 is straightforward and can be found in Appendix B.1.

LEMMA 5.1. *The opening tags in* $\mathrm{XML}(t)$ *are in the same order as the opening tags in* $\mathrm{XML}(\mathrm{FCNS}(t))$.

Since due to Lemma 5.1 the subsequences of opening tags in $\mathrm{XML}(t)$ and $\mathrm{XML}(\mathrm{FCNS}(t))$ coincide, we extract the subsequence of opening tags of $\mathrm{XML}(t)$, and we annotate them with left or right as they should be in $\mathrm{XML}(\mathrm{FCNS}(t))$. Recall that an opening tag is left if it is the opening tag of a first child, otherwise it is right. Furthermore, for later use we annotate each opening tag c with $\mathrm{depth}(c)$ in t and the position in the stream $\mathrm{pos}(c)$, see Algorithm 5.

Algorithm 5 Extracting the opening tags of $\mathrm{XML}(t)$

```
Require: input stream is a well-formed XML document
$d \leftarrow 0, p \leftarrow 0$
$D \leftarrow L$
while stream not empty do
    $x \leftarrow$ next tag on stream
    $p \leftarrow p + 1$
    if $x$ is an opening tag $c$ then
        $d \leftarrow d + 1$
        write on output stream $(c_D, d, p)$
        $D \leftarrow L$
    else {$x$ is a closing tag $\overline{c}$}
        $d \leftarrow d - 1$
        $D \leftarrow R$
    end if
end while
```

FACT 3. *Algorithm 5 is a streaming algorithm with space* $\mathrm{O}(\log N)$ *that, given* $\mathrm{XML}(t)$ *as input, outputs on an auxiliary stream the sequence of opening tags of* $\mathrm{XML}(\mathrm{FCNS}(t))$ *with left/right annotations, and furthermore, annotates each tag* c *with* $\mathrm{depth}(c)$ *and* $\mathrm{pos}(c)$. *It performs one read pass on the input stream and one write pass on the auxiliary stream.*

5.1.2. Computing the sequence of closing tags.

For a node v of some tree t, let $\mathrm{pos}'(v)$ and $\mathrm{pos}'(\bar{v})$ be the respective positions of the opening and closing tags of v in $\mathrm{XML}(\mathrm{FCNS}(t))$. Lemma 5.2 refers to the structure of the subsequence of closing tags in $\mathrm{XML}(\mathrm{FCNS}(t))$.

LEMMA 5.2. *Let* v_1, v_2 *be nodes of* t *with* $\mathrm{pos}(v_1) < \mathrm{pos}(v_2)$. *Then* $\mathrm{pos}'(\overline{v_2}) < \mathrm{pos}'(\overline{v_1})$ *iff:*

(*1*) v_2 *is in the subtree of* v_1 *in* t*;*
(*2*) *or* v_2 *is in the subtree of a right sibling of* v_2 *in* t.

PROOF. Suppose that either Item 1 or Item 2 is true. Note that for a node x, $\mathrm{XML}^{\mathrm{F}}(x)$ generates opening and closing tags for the entire subtree of x, and for all right siblings of x. Disregarding the annotations, we have $\mathrm{XML}^{\mathrm{F}}(v_2) =$

$v_2\mathrm{XML}^{\mathrm{F}}(\mathrm{fc}(v_2))\mathrm{XML}^{\mathrm{F}}(\mathrm{ns}(v_2))\overline{v_2}$, and hence $\overline{v_2}$ is preceded by all closing tags that are in the subtree of v_2 (Item 1) and all closing tags that are right siblings of v_2 or in the subtrees of right siblings of v_2 (Item 2).

We prove now that if Item 1 and Item 2 are false then $\mathrm{pos}'(\overline{v_1}) < \mathrm{pos}'(\overline{v_2})$. Suppose now that Item 1 and Item 2 are false. Let $p = \mathrm{lca}(v_1, v_2)$ where $\mathrm{lca}(x, y)$ denotes the lowest common ancestor of nodes x and y. Then $\mathrm{depth}(p) \leq \mathrm{depth}(v_1) - 2$ since otherwise Item 1 or Item 2 would be true.

Consider now $\mathrm{XML}^{\mathrm{F}}(p) = p\mathrm{XML}^{\mathrm{F}}(\mathrm{fc}(p))\mathrm{XML}^{\mathrm{F}}(\mathrm{ns}(p))\overline{p}$. If v_2 equals p then the lemma follows immediately since $\overline{v_1}$ is generated by $\mathrm{XML}^{\mathrm{F}}(\mathrm{fc}(p))$ and $\overline{p}$ is generated after $\mathrm{XML}^{\mathrm{F}}(\mathrm{fc}(p))$. Otherwise, let p' be the node at depth $\mathrm{depth}(p)+1$ that is on the path from v_1 to p. Then v_2 is a right sibling of p' or v_2 is in a subtree of a right sibling of p'. Consider $\mathrm{XML}^{\mathrm{F}}(p') = p'\mathrm{XML}^{\mathrm{F}}(\mathrm{fc}(p'))\mathrm{XML}^{\mathrm{F}}(\mathrm{ns}(p'))\overline{p'}$. Then v_1 is generated by $\mathrm{XML}^{\mathrm{F}}(\mathrm{fc}(p'))$ and v_2 is generated by $\mathrm{XML}^{\mathrm{F}}(\mathrm{ns}(p'))$ and this proves that $\mathrm{pos}'(\overline{v_1}) < \mathrm{pos}'(\overline{v_2})$. □

For computing the sequence of closing tags, we start with the sequence of opening tags of $\mathrm{XML}(\mathrm{FCNS}(t))$ as produced by the output of the Algorithm 5, that is, correctly annotated with left/right and with depth and position annotations. To obtain the correct subsequence of closing tags as in $\mathrm{XML}(\mathrm{FCNS}(t))$, we interpret the opening tags as closing tags and we sort them with a merge sort algorithm. Merge sort can be implemented as a streaming algorithm with $\mathrm{O}(\log(N))$ passes and 3 auxiliary streams [Grohe et al. 2009]. For the sake of simplicity, Algorithm 6 assumes an input of length 2^l for some $l > 0$.

Algorithm 6 Merge sort

```
Require: unsorted data of length $2^l$ on stream 1
1: for $i = 0 \ldots l-1$ do
2:    copy data in blocks of length $2^i$ from stream 1 alternately onto stream 2 and stream 3
3:    for $j = 1 \ldots 2^{l-i-1}$ do
4:       merge($2^i$)
5:    end for
6: end for
```

The function merge(b) reads simultaneously the next b values from stream 2 and stream 3, and merges them onto stream 1. The for loop in Line 3 of Algorithm 6 requires one read pass on stream 2, one read pass on stream 3, and one write pass on stream 1. See Figure 14 for an illustration.

	line 2 (copy)	line 3 (merge)
str 1:	B_1 B_2 B_3 B_4 $\cdots$ $B_{2^{l-i}}$	B_{12} B_{34} $\cdots$ $B_{2^{l-i-1}2^{l-i}}$
str 2:	B_1 B_3 $\cdots B_{l-i-1}$	B_1 B_3 $\cdots B_{l-i-1}$
str 3:	B_2 B_4 $\cdots$ B_{l-i}	B_2 B_4 $\cdots$ B_{l-i}

Fig. 14. Left: Illustration of the copy operation in Line 2 of Algorithm 6. Blocks from stream 1 are copied alternately onto stream 2 and stream 3. Right: Illustration of the merge operations executed within the for loop of Line 3 of Algorithm 6. The B_i are sorted blocks. All blocks B_i and B_{i+1} are merged into a sorted block $B_{i(i+1)}$.

In order to use merge sort, we have to define a comparator function that, given two closing tags $\overline{c_1}, \overline{c_2}$ with $\mathrm{pos}(c_1) < \mathrm{pos}(c_2)$, decides whether $\mathrm{pos}'(\overline{c_1}) < \mathrm{pos}'(\overline{c_2})$. Lemma 5.2 states that if c_2 is in the subtree of c_1 or c_2 is in the subtree of a right sibling of c_1 then $\mathrm{pos}'(\overline{c_2}) < \mathrm{pos}'(\overline{c_1})$, otherwise $\mathrm{pos}'(\overline{c_2}) > \mathrm{pos}'(\overline{c_1})$. Therefore, a comparator has to be able to distinguish between these two situations. This, however, seems difficult in the streaming model.

To overcome this problem, instead of only defining a comparison function, we design a complete merge function in Lemma 5.3 that, by construction, never encounters the situation that nodes v_1, v_2 with $\mathrm{pos}(v_1) < \mathrm{pos}(v_2)$ that do not fulfill Item 1 and Item 2 of Lemma 5.2 are compared. The key idea is to introduce *separator* tags which we denote by new tags outside of Σ. They are initially inserted right after each closing tag of a last child u. We denote by $\overline{\overline{u}}$ the separator we introduce when seeing the last child u, and we define $\mathrm{depth}(\overline{\overline{u}}) = \mathrm{depth}(u)$.

Algorithm 7 Unsorted sequence of closing tags of $\mathrm{XML}(\mathrm{FCNS}(t))$ with separators

```
Require: input stream is a well-formed XML document
$d \leftarrow 0, p \leftarrow 0$
$D \leftarrow L$
while stream not empty do
    $x \leftarrow$ next tag on stream
    $p \leftarrow p + 1$
    if $x$ is an opening tag $c$ then
        $d \leftarrow d + 1$
        write on output stream $(\overline{c_D}, d, p)$
        $D \leftarrow L$
    else {$x$ is a closing tag $\overline{c}$}
        if next item on stream is a closing tag then
            write on output stream $(\overline{\overline{c}}, d, p)$
        end if
        $d \leftarrow d - 1$
        $D \leftarrow R$
    end if
end while
```

FACT 4. *Algorithm 7 is a streaming algorithm with space* $\mathrm{O}(\log N)$ *that, given a sequence* $\mathrm{XML}(t)$ *on a stream, computes on an auxiliary stream the sequence of closing tags* $\mathrm{XML}(\mathrm{FCNS}(t))$ *together with their separators and annotates the tags with* depth, pos, *and left/right. It performs one read pass on the input stream and one write pass on the auxiliary stream.*

We have to define the way we integrate the separators into our sorting. Let $v_1, v_2, \ldots, v_k$ be the ordered sequence of the children of some node. For the separator $\overline{\overline{v_k}}$ we ask their position among the closing tags to satisfy for each node v:

$$\mathrm{pos}'(\overline{v}) < \mathrm{pos}'(\overline{\overline{v_k}}) \quad \text{iff} \quad \mathrm{pos}'(\overline{v}) \leq \mathrm{pos}'(\overline{v_1}); \tag{4}$$

and for any other separator $\overline{\overline{w_k}}$:

$$\mathrm{pos}'(\overline{\overline{v_k}}) < \mathrm{pos}'(\overline{\overline{w_k}}) \quad \text{iff} \quad \mathrm{pos}'(v_k) < \mathrm{pos}'(w_k). \tag{5}$$

Blocks appearing in merge sort fulfill a property that we call *well-sorted*. A block B of closing tags is *well-sorted* if the corresponding tags in $\mathrm{XML}(\mathrm{FCNS}(t))$ appear in the same order, and for all $\overline{v_1}, \overline{v_2} \in B$ with $\mathrm{pos}(v_1) < \mathrm{pos}(v_2)$, all closing tags $\overline{v}$ of nodes v with $\mathrm{pos}(v_1) < \mathrm{pos}(v) < \mathrm{pos}(v_2)$ are in B as well.

In addition, for two blocks B_1, B_2 of closing tags, we say that (B_1, B_2) is a *well-sorted adjacent pair*, if B_1 and B_2 are well-sorted, for each closing tag $\overline{v_1} \in B_1$ and each closing tag $\overline{v_2} \in B_2$, $\mathrm{pos}(v_1) < \mathrm{pos}(v_2)$ is satisfied, and furthermore, all closing tags $\overline{v}$ of nodes v with $\mathrm{pos}(v_1) < \mathrm{pos}(v) < \mathrm{pos}(v_2)$ are either in B_1 or B_2.

The following lemma shows that we can merge a well-sorted adjacent pair correctly.

LEMMA 5.3. *Let (B_1, B_2) be a well-sorted adjacent pair, and let $b_1 = B_1[p_1]$ and $b_2 = B_2[p_2]$ for some p_1, p_2. Assume that $\mathrm{pos}'(b) < \mathrm{pos}'(b_1)$ and $\mathrm{pos}'(b) < \mathrm{pos}'(b_2)$, for all $b \in B_1[1, p_1 - 1] \cup B_2[1, p_2 - 1]$. Then:*

(*1*) *If b_1 is a separator, or there is a separator in B_1 after b_1, then $\mathrm{pos}'(b_1) < \mathrm{pos}'(b_2)$;*
(*2*) *Else if b_2 is a separator then:*
  (*a*) *if $\mathrm{depth}(b_1) < \mathrm{depth}(b_2)$ then $\mathrm{pos}'(b_2) < \mathrm{pos}'(b_1)$,*
  (*b*) *else $\mathrm{depth}(b_1) = \mathrm{depth}(b_2)$ and $\mathrm{pos}'(b_1) < \mathrm{pos}'(b_2)$;*
(*3*) *Else (neither b_1 nor b_2 are separators and there is no separator in B_1 after b_1): $\mathrm{pos}'(b_2) < \mathrm{pos}'(b_1)$.*

PROOF. Let (B_1, B_2) be a well-sorted adjacent pair. Let $l = \max\{i : B_1[i] \text{ is a separator}\}$. If there are no separators in B_1, let $l = 0$.

Item 1. Since B_1 is well-sorted, we only need to check that $\mathrm{pos}'(B_1\,[l]) < \mathrm{pos}'(B_2\,[1])$. Denote by u the last child that was responsible for the insertion of the separator tag $B_1[l]$. Let u' be the left-most sibling of u. Due to Equation (4) it suffices to show that $\mathrm{pos}'(\overline{u'}) < \mathrm{pos}'(B_2[1])$. Since the separator $B_1[l]$ indicates that the last child u has been seen, $B_2[1]$ is not in the subtree of u' or in a subtree of a right sibling of u'. Therefore, by Lemma 5.2 we get $\mathrm{pos}'(\overline{u'}) < \mathrm{pos}'(B_2[1])$.

Item 2. Let v denote a node in the tree with children $v_1, \ldots, v_k$. First, note that the separator $\overline{\overline{v_k}}$ is initially inserted after v_k. Furthermore, between the initial position of any v_i and $\overline{\overline{v_k}}$ there are no other separators with a depth smaller than $\mathrm{depth}(\overline{\overline{v_k}})$. Therefore, it can not happen that the node b_1 is compared to a separator tag with depth smaller than $\mathrm{depth}(b_1)$.

If $\mathrm{depth}(b_2) = \mathrm{depth}(b_1)$ then b_2 is the seperator tag that was inserted after the right-most sibling of b_1. Let l be the left-most sibling of b_1. Then $\mathrm{pos}'(b_1) < \mathrm{pos}'(l)$ and therefore by Equation 4 we have $\mathrm{pos}'(b_1) < \mathrm{pos}'(b_2)$. If $\mathrm{depth}(b_2) > \mathrm{depth}(b_1)$ then b_2 is the separator that was introduced after a node that is either in the subtree of b_1 or in the subtree of a right sibling of b_1. Let l' denote the left-most sibling of that node. By Lemma 5.2 we have $\mathrm{pos}'(l') < \mathrm{pos}'(b_1)$ and hence by Equation 4 we have $\mathrm{pos}'(b_2) < \mathrm{pos}'(b_1)$.

Item 3. We argue that b_2 is in the subtree of b_1 or b_2 is in the subtree of a right sibling of b_1. Then, by Lemma 5.2, we have $\mathrm{pos}'(b_2) < \mathrm{pos}'(b_1)$. Suppose for the sake of contradiction that this is not the case. Then the separator that was introduced after the right-most sibling of b_1 must be in $B_1[p_1 + 1, k] \cup B_2[1, p_2 - 1]$, where $k = |B_1|$. Suppose that this separator was in $B_1[p_1 + 1, k]$. Then this is a contradiction since this case is treated in Item 1 of this lemma. Suppose that this separator was in $B_2[1, p_2 - 1]$. Then this is a contradiction to the assumption of the lemma that $\mathrm{pos}'(B_2[j]) < \mathrm{pos}'(b_1)$ for all $j < p_1$. □

LEMMA 5.4. *There is a $\mathrm{O}(\log N)$-pass streaming algorithm with space $\mathrm{O}(\log N)$ and 3 auxiliary streams that computes the subsequence of closing tags of the FCNS encoding of any XML document given in the input stream.*

PROOF. Using Algorithm 7, we compute on the first auxiliary stream the sequence of opening tags interpreted as closing tags with corresponding annotations, together with separators.

We show that we can do a merge sort algorithm with a merge function inspired by Lemma 5.3 on the first three auxiliary streams with $\mathrm{O}(\log N)$ space and passes. For that assume that the first stream contains a sequence $(B_1, B_2, \ldots, B_M)$ of blocks of size 2^i. For simplicity we assume that M is even, otherwise we add an empty block. We alternately copy odd blocks on the second stream, and even blocks on the third stream. For a block B_{2i} that we write on the third stream, we write before each of them, the number of separators that occur in the block B_{2i-1} that was copied on the second stream.

Then we merge sequentially all pairs of blocks (B_{2k-1}, B_{2k}) for $1 \leq k \leq M/2$ using Lemma 5.3. Note that $(B_{2k-1}, B_{2k})_k$ are all well-sorted pairs. Let $l = \max\{i : B_{2k-1}[i]$ is a separator$\}$. Firstly, we copy elements $B_{2k-1}[1, l]$ onto auxiliary stream 1. Knowing the number of separators in B_{2k-1} allows us to perform this operation. The correctness of this step follows from Item 1 of Lemma 5.3. Then, we merge blocks $B_{2k-1}[l+1, 2^i]$ and B_{2k} by using the comparison function defined in Items 2 and 3 of Lemma 5.3. □

5.1.3. Merging opening and closing tags. Merging the subsequence of opening tags of $\mathrm{XML}(\mathrm{FCNS}(t))$ and the subsequence of closing tags of $\mathrm{XML}(\mathrm{FCNS}(t))$ can be done by simultaneously reading the two subsequences and performing one write pass over an auxiliary stream.

Algorithm 8 Merging the sequence of opening and closing tags

Require:

— stream 1: annotated opening tags as output by Algorithm 5

— stream 2: annotated closing tags as output by the algorithm of Lemma 7

```
while stream 2 not empty do
    $(c_1, p_1, d_1), \ldots, (c_k, p_k, d_k)(c_{k+1}, p_{k+1}, d_{k+1}) \leftarrow$ next $k+1$ annotated opening tags from stream 1 such that $d_1 \leq d_2 \leq \cdots \leq d_k$ and $d_k > d_{k+1}$ and $(c_{k+1}, p_{k+1}, d_{k+1})$ is not discarded from the stream
    output $c_1 \ldots c_k$ on output stream
    for $i = 1 \ldots d_k - d_{k+1}$ do
        $(\overline{c_1}, p_1, d_1), \ldots, (\overline{c_l}, p_l, d_l)(\overline{\overline{c_{l+1}}}, p_{l+1}, d_{l+1}) \leftarrow$ next $l$ annotated closing tags from stream 2 such that $(\overline{c_i})_{1 \leq i \leq l}$ are closing tags and $\overline{\overline{c_{l+1}}}$ is a separator
        output $\overline{c_1} \ldots \overline{c_l}$ on output stream
    end for
end while
```

When merging the sequence of opening tags and closing tags, we have to write closing tags only between two opening tags a_m, b_{i+1} (see Figure 15) if $\mathrm{depth}(a_m) > \mathrm{depth}(b_{i+1})$ in t. Figure 15 shows the closing tags that have to be written at that moment. The sequences $\overline{a_m} \ldots \overline{a_1}$, $\overline{e}$, $\overline{d_l} \ldots \overline{d_1}$ and $\overline{c_j} \ldots \overline{c_1}$ are all separated by a separator in the sequence of closing tags. Therefore, it is enough to write the next $\mathrm{depth}(a_m) - \mathrm{depth}(b_{i+1})$ blocks of closing tags that are separated by a separator between a_m and b_{i+1}.

LEMMA 5.5. *Algorithm 8 merges correctly the sequence of opening tags and closing tags using space* $\mathrm{O}(\log N)$.

PROOF. First, we argue that closing tags only have to be written between consecutive opening tags a and b in the sequence of opening tags such that $\mathrm{depth}(a) > \mathrm{depth}(b)$. We have $\mathrm{XML}^F(a) = a\mathrm{XML}^F(\mathrm{fc}(a))\mathrm{XML}^F(\mathrm{ns}(a))\overline{a}$, and therefore if $\mathrm{depth}(a) < \mathrm{depth}(b)$ then b is either the first child of a or if a is a leaf then b is the next sibling of a. In both cases, there are no closing tags between a and b.

Figure 15 illustrates the closing tags that have to be written beween two consecutive opening tags a and b with $\mathrm{depth}(a) > \mathrm{depth}(b)$. Since closing tags at different levels are separated by a separator, it is enough to write the next $\mathrm{depth}(a) - \mathrm{depth}(b)$ blocks of closing tags that are separated by a separator between the tags a and b. □

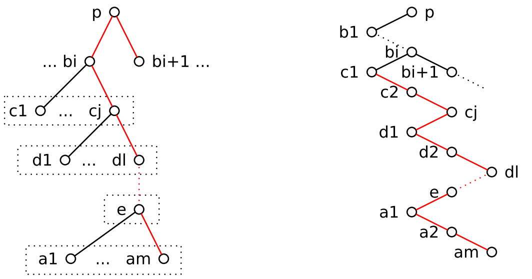


Fig. 15. A part of a tree and its FCNS encoding. Consider nodes a_m and b_{i+1}. In the FCNS encoding, the sequence of closing tags $\overline{a_m} \dots \overline{a_1}\overline{e}\overline{d_l} \dots \overline{d_1}\overline{c_j} \dots \overline{c_1}$ has to be written in between the opening tag a_m and b_{i+1}.

From Fact 3, Lemma 5.4 and Lemma 5.5 we obtain Theorem 5.6.

THEOREM 5.6. *There is a* $\mathrm{O}(\log N)$*-pass streaming algorithm with space* $\mathrm{O}(\log N)$ *and* 3 *auxiliary streams and* $\mathrm{O}(1)$ *processing time per letter that computes on the third auxiliary stream the FCNS transformation of any XML document given in the input stream.*

PROOF. Firstly, we compute according to Lemma 5.4 the sequence of closing tags and we store them on auxiliary stream 1. Then, by Fact 3 we extract the sequence of opening tags, and we store them on auxiliary stream 2. By Lemma 5.5 we can merge the tags of auxiliary stream 1 and auxiliary stream 2 correctly onto stream 3.

The space requirements of these operations do not exceed $\mathrm{O}(\log N)$. The processing time per letter of these operations is constant. □

The algorithm described in the proof of Theorem 5.6 can be easily modified such that it outputs $\mathrm{XML}(\mathrm{FCNS}^{\perp}(t))$ instead of $\mathrm{XML}(\mathrm{FCNS}(t))$. We state this fact in the following.

COROLLARY 5.7. *There is a* $\mathrm{O}(\log N)$*-pass streaming algorithm with space* $\mathrm{O}(\log N)$ *and* 3 *auxiliary streams and* $\mathrm{O}(1)$ *processing time per letter that computes on the third auxiliary stream the* $\mathrm{FCNS}^{\perp}$ *encoding of any XML document given in the input stream.*

PROOF. Firstly, we use the algorithm described in Theorem 5.6 to compute the transformation $\mathrm{XML}(t)$ into $\mathrm{XML}(\mathrm{FCNS}(t))$. Then, with an additional read pass and an additional write pass we transform $\mathrm{XML}(\mathrm{FCNS}(t))$ into $\mathrm{XML}(\mathrm{FCNS}^{\perp}(t))$. To perform this transformation, we read the tags of $\mathrm{XML}(\mathrm{FCNS}(t))$ and output them on another stream without left/right annotations, and at the same time we insert leaves labeled with $\perp$. Such a leaf has to be inserted below internal nodes that have only a single child. The left/right annotations of the input stream allow us to recognize those nodes. Note that the transformation $\mathrm{XML}(\mathrm{FCNS}(t))$ into $\mathrm{XML}(\mathrm{FCNS}^{\perp}(t))$ requires only constant space. □

5.2. Checking Validity on the encoded form

The problem of validating trees given in their encoded form and the problem of validating binary trees are similar. We will provide intuition that basically *any* streaming algorithm that decides validity of binary trees by calling a check function upon all triplets (v, v_1, v_2) of internal nodes v with children v_1, v_2 ((v, ϵ, ϵ) for leaves) can be transformed into an algorithm that decides validity of trees given in their encoded form. We will explicitly show how to use the bidirectional 2-pass algorithm, Algorithm 3, and the one-pass algorithm, Algorithm 2, to perform this task.

To validate a node v with children $v_1, \ldots, v_k$, an algorithm has to ensure that the sequence $v_1 \ldots v_k$ is valid with respect to the regular expression $d(v)$. To perform such a check, an algorithm has to gather the relevant information, which is the label of v and the label of its children $v_1, \ldots, v_k$, from the stream. Figure 16 illustrates the fact that $\overline{v_k} \ldots \overline{v_1}$ forms a substring in $\mathrm{XML}(\mathrm{FCNS}^{\perp}(t))$. Suppose that the information about the labels of the children $v_1, \ldots, v_k$ was available at node v_1 in $\mathrm{FCNS}^{\perp}(t)$ (in a compressed form since the number of children of a node can be large). Then we could use any algorithm validating binary trees which uses a check function as described above for our purpose: since such an algorithm relates a node to its two children, we can use this algorithm on $\mathrm{FCNS}^{\perp}(t)$ to relate a node to its left child.

Granting access to all children labels when it is required is established with the help of a finite automaton that we discuss later. Consider a left-to-right pass over $\mathrm{FCNS}^{\perp}(t)$. When seeing the sequence $\overline{v_k} \ldots \overline{v_1}$, we feed it into a finite automaton. The resulting state is a compressed version of this sequence. A binary tree validity algorithm will then relate this state to the parent node. The details follow.

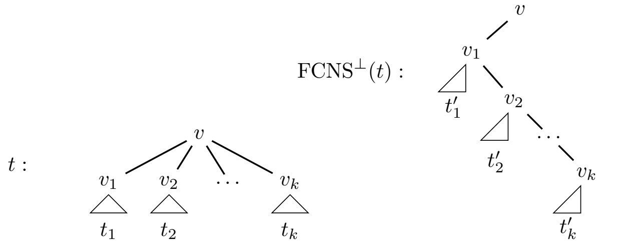


Fig. 16. A tree t and its $\mathrm{FCNS}^{\perp}$ encoding. While the opening and closing tags of the children of a node v are separated by the subtrees $t_1, \ldots t_k$ in $\mathrm{XML}(t)$, the closing tags of the children of v are consecutive in $\mathrm{XML}^{\mathrm{F}^{\perp}}(t)$ in reverse order, that is $\overline{v_k}\,\overline{v_{k-1}} \ldots \overline{v_2}\,\overline{v_1}$ is a substring of $\mathrm{XML}^{\mathrm{F}^{\perp}}(t)$.

For a non-leaf node v, we gather the information of the children nodes $v_1, \ldots, v_k$ with the help of finite automata $\mathcal{A}_1$ (for left-to-right passes) and $\mathcal{A}_2$ (for right-to-left passes).

We denote by $(\Sigma, Q, q_0, \delta, F)$ a deterministic finite automaton where Σ is its input alphabet, Q is the state set, q_0 is its initial state, $\delta : Q \times \Sigma \to Q$ is the transition function, and F is a set of final states. Furthermore, for a word $\omega = \omega_1 \ldots \omega_n$ of length n, we define ω^{rev} to be ω read from right to left, that is $\omega^{\mathrm{rev}} = \omega_n \ldots \omega_1$.

LEMMA 5.8. *Let $D = (\Sigma, d, s_d)$ denote a DTD. Then there is a deterministic finite automaton $\mathcal{A}_1 = (\Sigma, Q_1, q_0^1, \delta_1, F_1)$ that for any $v \in \Sigma$ and any $v_1 \ldots v_k$ in Σ^k accepts the word $v_k \ldots v_1 v$ only if $v_1 \ldots v_k$ fulfills the regular expression $d(v)$.*

PROOF. For $a \in \Sigma$, denote by A_a a deterministic finite automaton that accepts the regular expression $d(a)$. We compose the A_a as in the left illustration of Figure 17 to an automaton A that accepts words ω' such that $\omega' = a\omega$, $a \in \Sigma, \omega \in \Sigma^*$ if $\omega \in d(a)$. $\mathcal{A}_1$ is a deterministic finite automaton that accepts a word ω, iff ω^{rev} is accepted by A. □

LEMMA 5.9. *Let $D = (\Sigma, d, s_d)$ denote a DTD. Then there is a deterministic finite automaton $\mathcal{A}_2 = (\Sigma, Q_2, q_0^2, \delta_2, F_2)$ that for any $v \in \Sigma$ and any $v_1 \ldots v_k$ in Σ^k accepts the word $v_1 \ldots v_k v$ only if $v_1 \ldots v_k$ fulfills the regular expression $d(v)$.*

PROOF. For $a \in \Sigma$, denote by A_a a deterministic finite automaton that accepts the regular expression $d(a)$. We compose the A_a as in the right illustration of Figure 17 to an automaton A that accepts words ω' such that $\omega' = \omega a$, $a \in \Sigma, \omega \in \Sigma^*$ if $\omega \in d(a)$. Then $\mathcal{A}_2$ is a deterministic version of A without ϵ transitions. □

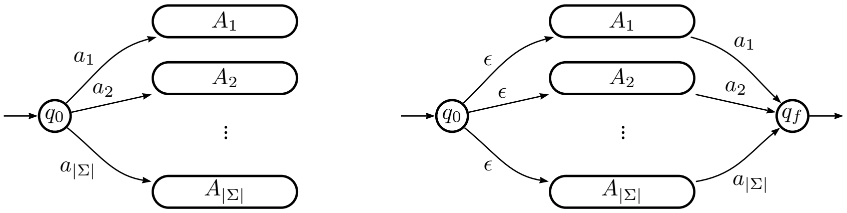


Fig. 17. Left: Automaton A. $\mathcal{A}_1$ accepts words ω if A accepts ω^{rev}. Right: Automaton $\mathcal{A}_2$ is a version of the illustrated automaton without ϵ transitions.

We show now that by the help of automata $\mathcal{A}_1$ and $\mathcal{A}_2$, Algorithm 3 can be reused for the validation of trees given in their encoded form.

THEOREM 5.10. *There is a bidirectional two-pass deterministic algorithm for* VALIDITY *with space* $\mathrm{O}(\log^2 N)$ *and* $\mathrm{O}(\log N)$ *processing time per letter when the input is given in its FCNS encoding.*

PROOF. We run a modified version of Algorithm 3 on $\mathrm{XML}(\mathrm{FCNS}^{\perp}(t))$. The modifications concern the subroutine described in Algorithm 4. The modifications are different for the left-to-right pass and the right-to-left pass.

Firstly, we consider the left-to-right pass. We will annotate the closing tags of left children on the fly by states of the automaton $\mathcal{A}_1$ as described in Lemma 5.8. Let $v_1, \dots, v_k$ denote the children of a node v. Then the annotation of $\overline{v_1}$ is a state that we denote by $q_1(v)$. $q_1(v)$ is the resulting state of $\mathcal{A}_1$ when feeding the sequence $v_k, \dots, v_1$ into it. We describe later how to compute it on the fly. Given this annotation, we use a different implementation of the check function. For internal nodes v with first child v_1 and annotation $q_1(v)$, the check function simply computes the state $\delta_1(q_1(v), v)$ and stops if the prior state is not an accepting state. Note that by the definition of $\mathcal{A}_1$, v is valid if $\delta_1(q_1(v), v)$ is an accepting state.

We discuss now how to compute this annotation. As discussed before and illustrated in Figure 16, the closing tags $\overline{v_k} \dots \overline{v_1}$ of children $v_1, \dots, v_k$ of a node v form a substring. Hence, as soon as we see $\overline{v_k}$ which we can easily identify since it is a right leaf, we run the automaton $\mathcal{A}_1$ on the labels of the upcoming closing tags. We stop this procedure after $\overline{v_1}$ is read which we can identify since $\overline{v_1}$ is followed by an opening tag. Hence, when $\overline{v_1}$ is pushed on the stack (in Algorithm 4 it is actually pushed on the stack together with the opening tag of the right child of v), we can annotate it with $q_1(\overline{v_1})$.

Consider now a right-to-left pass. Note that in a right-to-left pass, closing tags are interpreted as opening tags and vice versa. This implies that a left child becomes a right child and a right child becomes a left child. Let $v_1, \dots, v_k$ denote the children of a node v. Then in a right-to-left pass, we see the sequence of opening tags $v_1, \dots, v_k$ as a substring, where v_1 is a right opening tag and $v_2, \dots, v_k$ are left opening tags. We will annotate the closing tag of the left child of v. Note that due to the exchange of the role of left and right, the left closing tag is the next sibling of v and not the first child. Since our input tree $\mathrm{FCNS}^{\perp}(t)$ is a binary tree, it is guaranteed that this node exists. The annotation is the state $q_2(v)$. $q_2(v)$ is obtained by feeding the sequence $v_1, \dots, v_k$ into the automaton $\mathcal{A}_2$, who is described in Lemma 5.9. The check function then computes $\delta_2(q_2(v), v)$ and stops if the resulting state is not an accepting state.

We discuss now that this annotation can be computed on the fly and it can be added correctly to the closing tag of the left child of v. The main difference to the left-to-right pass is that we compute the annotation after having pushed the children of v onto the stack and we add the annotation afterwards. Denote by v_l the left child of v in $\mathrm{FCNS}^{\perp}(t)$. Then in the right-to-left pass we see the substring $\overline{v_l}v_1v_2\ldots v_k$. Algorithm 4 pushes $\overline{v_l}, v_1$ on the stack as soon as v_1 is seen. We then feed the sequence $v_1v_2\ldots v_k$ into $\mathcal{A}_2$. As soon as v_k is read which can be easily identified since v_k is either a leaf of followed by a right opening tag, we annotate the left closing tag of the topmost stack item by the state $q_2(v)$.

Correctness, that is the validation of all nodes, follows then from the correctness of Algorithm 3. The automata $\mathcal{A}_1, \mathcal{A}_2$ are of constant size since we assumed that the input DTD is of constant size. Hence, the described algorithm has the same space complexity as Algorithm 3. □

By modifying the one-pass algorithm Algorithm 2 in a similar way, the following theorem can be obtained.

THEOREM 5.11. *There is a one-pass deterministic algorithm for* VALIDITY *with space* $\mathrm{O}(\sqrt{N \log N})$ *and* $\mathrm{O}(1)$ *processing time per letter when the input is given in its* $\mathrm{FCNS}^{\perp}$ *encoding.*

PROOF. We reuse Algorithm 2. Concerning the modifications, the idea is the same as for the left-to-right pass of the algorithm described in the proof of Theorem 5.10. For all internal nodes v with children $v_1, \ldots, v_k$, we compress the sequence $v_1, \ldots, v_k$ into a state $q_1(v)$ of the finite automaton $\mathcal{A}_1$ who is described in Lemma 5.8. $q_1(v)$ is obtained by feeding $v_k \ldots v_1$ into $\mathcal{A}_1$ which can be done since $\overline{v_k} \ldots \overline{v_1}$ forms a substring of the input XML sequence. We annotate the closing tag of v_1 with this state. The check routine is modified in the same way as in the proof of Theorem 5.10: only if $\delta_1(q_1(v), v)$ is an accepting state then v is valid, otherwise the check routine aborts and the algorithm reports an invalid node. The correctness of Algorithm 2 ensures the validation of all nodes. □

Applying the bidirectional algorithm of Theorem 5.10 on the encoded form $\mathrm{XML}(\mathrm{FCNS}^{\perp}(t))$, we obtain that validity of general trees can be decided memory efficiently in the streaming model with auxiliary streams.

COROLLARY 5.12. *There is a bidirectional* $\mathrm{O}(\log N)$*-pass deterministic streaming algorithm for* VALIDITY *with space* $\mathrm{O}(\log^2 N)$*,* $\mathrm{O}(\log N)$ *processing time per letter, and* 3 *auxiliary streams.*

PROOF. We perform the transformation $\mathrm{XML}(t)$ into $\mathrm{XML}(\mathrm{FCNS}^{\perp}(t))$ with the algorithm stated in Corollary 5.7. Then, we run the two-pass bidirectional algorithm of Theorem 5.10 on $\mathrm{XML}(\mathrm{FCNS}^{\perp}(t))$ and the result follows. □

Note that this result only holds for the validation of DTDs. Nothing is known about the validation of more powerful validity schemas such as extended DTDs or XML Schema if access to auxiliary streams is granted.

5.3. Decoding

In the following, we present streaming algorithms for FCNS decoding, that is, given $\mathrm{XML}(\mathrm{FCNS}(t))$ of some tree t, output $\mathrm{XML}(t)$. These results complement our results on the computation of the FCNS encoding and so may be primarily of theoretical interest. There are, however, potential applications: It may be advantageous to store the FCNS encoding of an XML file instead of the XML file itself. Then validity could be efficiently ensured by Algorithm 5.10 with two bidirectional passes and space $\mathrm{O}(\log^2 N)$. The original document could then be *exported* by means of the algorithms that we present in this section. The applicability of this approach is left open.

We start with a non-streaming algorithm, Algorithm 9 performing this task.

Algorithm 9 offline algorithm for FCNS decoding

```
1: for $i = 1 \to 2N$ do
2:   if $X[i]$ is an opening tag then
3:     write $X[i]$
4:     if $X[i]$ does not have a left subtree then {$X[i]$ is a leaf}
5:       write $\overline{X[i]}$
6:     end if
7:   else if $X[i]$ is a left closing tag then {See Figure 18}
8:     let $p$ be the parent node of $X[i]$
9:     write $\overline{p}$
10:  end if
11: end for
```

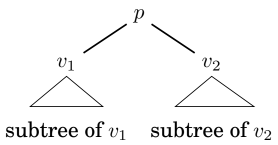


Fig. 18. The main difficulty of the FCNS decoding is to write the closing tag of a node p when the closing tag of its left child is seen. This is difficult when the subtrees of v_1 and v_2 are large.

We first discuss the correctness of Algorithm 9. We show that the algorithm run on $\mathrm{XML}(\mathrm{FCNS}(t))$ computes the function $\mathrm{dec}(\mathrm{root}(t))$ which we define in the following. Let t be a tree and let $x \in t$ be a node, then

$$\begin{aligned} \mathrm{dec}(x) &= x\,\mathrm{dec}(\mathrm{fc}(x))\,\overline{x}\,\mathrm{dec}(\mathrm{ns}(x)), \\ \mathrm{dec}(\perp) &= \epsilon. \end{aligned}$$

The only difference between dec and $\mathrm{XML}^{\mathrm{F}}$ is that for some non-leaf node x, $\mathrm{dec}(x)$ outputs $\overline{x}$ between the recursive calls to $\mathrm{dec}(\mathrm{fc}(x))$ and $\mathrm{dec}(\mathrm{ns}(x))$ while $\mathrm{XML}^{\mathrm{F}}$ outputs $\overline{x}$ at the very end. Algorithm 9 computes dec since it ignores the closing tags of the FCNS encoding and it inserts closing tags when we do a transition from the left child to a right child, that is between the recursive calls to $\mathrm{dec}(\mathrm{fc}(x))$ and $\mathrm{dec}(\mathrm{ns}(x))$. We show in Lemma 5.13 that $\mathrm{dec}(\mathrm{root}(t))$ produces the same output as $\mathrm{XML}(\mathrm{root}(t))$. The proof can be found in Appendix B.2.

LEMMA 5.13. $\mathrm{dec}(\mathrm{root}(t)) = \mathrm{XML}(\mathrm{root}(t))$.

COROLLARY 5.14. *Algorithm 9 is an offline algorithm that computes* $\mathrm{XML}(t)$ *given* $\mathrm{XML}(\mathrm{FCNS}(t))$.

We describe how this algorithm can be converted into a streaming algorithm. For an opening tag $X[i]$, checking the condition in Line 4 can easily be done by investigating $X[i+1]$. If $X[i+1]$ is a right opening tag or equals $\overline{X[i]}$, $X[i]$ does not have a left subtree. The difficulty in converting this algorithm into a streaming algorithm is in Line 8, it is difficult to keep track of opening tags until the respective closing tags of their left children are seen, and indeed, this cannot be done with sublinear space in one pass, see Theorem 6.2.

In the following, we present a streaming algorithm that performs one pass over the input, but two passes over the output, and uses $O(\sqrt{N \log N})$ space, and a streaming algorithm that performs $\mathrm{O}(\log N)$ passes over the input and 3 auxiliary streams using $\mathrm{O}(\log^2(N))$ space.

5.3.1. One read-pass and two write-passes. We read blocks of size $\sqrt{N \log N}$ and execute Algorithm 9 on that block. In Lemma 4.1 we showed that in any block there is at most one left closing tag for which the parent's opening and closing tag are not in that block. Hence per block there is at most one left closing tag for which we can not obtain the label of the parent node. We call this closing tag *critical*. In this case we write a *dummy symbol* on the output stream that will be overwritten by the parent's closing tag in the second pass. The closing tag of the parent node will arrive in a subsequent block, and it can easily be identified as this since it is the next closing tag arriving at a depth -1 of the critical closing tag. We store it upon its arrival in our random access memory. Since there is at most one critical closing tag per block and we have a block size of $\sqrt{N \log N}$, we have to recover at most $\mathrm{O}(\sqrt{N / \log N})$ parent nodes. At the end of the pass over the input stream we have recovered all closing tags of parent nodes for which we wrote dummy symbols on the output stream. In a second pass over the output stream we overwrite the dummy symbols by the correct closing tags.

The space complexity uses Lemma 4.1 that was already applied in Section 4.1.

THEOREM 5.15. *There is a streaming algorithm using* $\mathrm{O}(\sqrt{N \log N})$ *space and* $\mathrm{O}(1)$ *processing time per letter which performs one pass over the input stream containing* $\mathrm{XML}(t)$ *and two passes over the output stream onto which it outputs* $\mathrm{XML}(\mathrm{FCNS}(t))$.

5.3.2. Logarithmic number of passes. Again, we use the offline Algorithm 9 as a starting point for the algorithm we design now. For coping with the problem that it is hard to remember all opening parent tags when their corresponding closing tag ought to be written on the output, we always write *dummy symbols* on the output stream for all parent closing tags. The crux then is the following observation:

FACT 5. *Let* $\overline{c_1}_{\mathrm{L}} \dots \overline{c_N}_{\mathrm{L}}$ *be the subsequence of closing tags of left children of* $\mathrm{XML}(\mathrm{FCNS}(t))$. *Then the sequence* $\overline{p_1} \dots \overline{p_N}$ *is a subsequence of* $\mathrm{XML}(t)$ *where* p_i *is the parent node of* c_i *in* $\mathrm{FCNS}(t)$.

We apply a modified version of our bidirectional two-pass Algorithm 3 to recover the missing tags. Instead of checking validity, once the check function is called in Algorithm 4 with variables (a, b, c), we output the parent label a onto an auxiliary stream, annotated with $\mathrm{pos}(b)$. We do the same in a reverse pass over the input stream counting positions from $2N$ downwards to 1. In so doing, the auxiliary stream contains all parent labels for which dummy symbols are written on the output stream.

Fact 5 shows that it is enough to sort by means of two further auxiliary streams the auxiliary stream with respect to the annotated position of the closing tags of the left children of these nodes. In a last pass we insert the parent closing tags into the output stream.

THEOREM 5.16. *There is a* $\mathrm{O}(\log N)$*-pass streaming algorithm with space* $\mathrm{O}(\log^2 N)$ *and* $\mathrm{O}(\log N)$ *processing time per letter and* 3 *auxiliary streams that computes on the third auxiliary stream the FCNS decoding of any FCNS encoded document given in the input stream.*

6. LOWER BOUNDS FOR FCNS ENCODING AND DECODING

6.1. Lower bound for FCNS encoding

Let $x \in \Sigma^n$. We define a family of hard instances $t(x)$ of length $N = \Theta(n)$ for the computation of $\mathrm{XML}(\mathrm{FCNS}(t(x)))$ given $\mathrm{XML}(t(x))$ as in Figure 19.

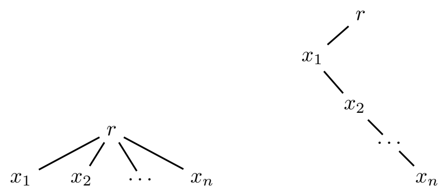


Fig. 19. Left: hard instance. Right: its FCNS encoded form.

It is easy to see that computing the sequence of closing tags in the FCNS encoding requires to reverse a stream. Let t be a hard instance. Then $\mathrm{XML}(t) = r x_1 \overline{x_1} x_2 \overline{x_2} \dots x_n \overline{x_n} \overline{r}$, and $\mathrm{XML}(\mathrm{FCNS}(t)) = r_{\mathrm{L}} x_{1\mathrm{L}} x_{2\mathrm{R}} \dots x_{n\mathrm{R}} \overline{x_{n\mathrm{R}}}\,\overline{x_{n-1\mathrm{R}}} \dots \overline{x_{2\mathrm{R}}}\,\overline{x_{1\mathrm{L}}}\,\overline{r_{\mathrm{L}}}$. Since writing the closing tags on the output stream can only start after reading x_n, we deduce that memory space $\Omega(n)$ is required in order to store all previous tags $x_1, \dots, x_{n-1}$.

FACT 6. *Every randomized streaming algorithm for FCNS encoding that performs one pass on the input stream and one pass on the output stream with bounded error requires $\Omega(N)$ space.*

6.2. Lower bound for FCNS decoding

We define now a family of hard instances of length $N = \Theta(n)$ for decoding a FCNS encoded tree. Let $X \in \{0,1\}^n, Y \in \{0,1\}^n$ and $K \in [n]$ be uniformly distributed random variables. Let $x \leftarrow X, y \leftarrow Y$ and $k \leftarrow K$. Denote by $t'(y)$ an arbitrary but fixed two-ranked tree with n nodes that are labeled by $y_1, \dots, y_n$, and let $s'(y)$ be the decoded form of $t'(y)$. We define then the hard instance $t(x, y, k)$ and its decoded form $s(x, y, k)$ as in Figure 20.

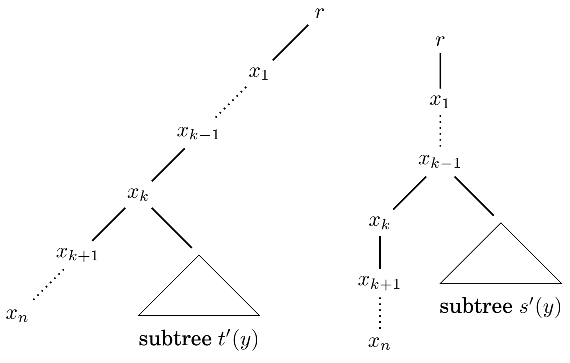


Fig. 20. Left: hard instance $t(x, y, k)$ in FCNS form. Right: its decoded form $s(x, y, k)$. $s'(y)$ is the decoded form of subtree $t'(y)$.

Then we have

$$\begin{aligned} \mathrm{XML}(t(x,y,k)) &= rx_{1\mathrm{L}} \dots x_{n\mathrm{L}}\overline{x_{n\mathrm{L}}} \dots \overline{x_{k+1\mathrm{L}}}\mathrm{XML}(t'(y))\overline{x_{k\mathrm{L}}} \dots \overline{x_{1\mathrm{L}}r_{\mathrm{L}}}, \text{ and} \\ \mathrm{XML}(s(x,y,k)) &= rx_1 \dots x_n\overline{x_n} \dots \overline{x_k}\mathrm{XML}(s'(y))\overline{x_{k-1}} \dots \overline{x_1 r}. \end{aligned}$$

The crucial difference between $t(x,y,k)$ and $s(x,y,k)$ is that the subtree $t'(y)$ is attached to node x_k in $t(x,y,k)$ while the subtree $s'(y)$ is attached to node x_{k-1} in $s(x,y,k)$. As a consequence, $\mathrm{XML}(t'(y))\overline{x_{k\mathrm{L}}}$ is a substring of $\mathrm{XML}(t(x,y,k))$ while $\overline{x_k}\mathrm{XML}(s'(y))$ is a substring of $\mathrm{XML}(s(x,y,k))$. We will see that it is difficult to write the substring $\overline{x_k}\mathrm{XML}(s'(y))$ with sublinear space.

Note that $\overline{x_k}$ has to be written before $s'(y)$ (which is the decoded version of $t'(y)$). However, $\overline{x_k}$ appears after the subtree $t'(y)$ in $\mathrm{XML}(t(x,y,k))$. Hence, we either have to store the entire subtree $t'(y)$ in memory using $\Theta(n)$ space, or we have to infer $\overline{x_k}$ from the opening tag x_{kL} in $t(x,y,k)$. Since, however, k is not known to the algorithm before seeing the subtree $t'(y)$, this can not be done with sublinear memory.

We start with the definition of a one-way three-party communication game that we denote by INDEXCOPY. Let the input be uniformly distributed random variables $X \in \{0,1\}^n, Y \in \{0,1\}^n$ and $K \in [n]$. They are given to the three parties Alice, Bob and Charlie as follows:

$$\begin{array}{ccccc} \text{Alice} & \xrightarrow{M_A} & \text{Bob} & \xrightarrow{M_B} & \text{Charlie} \\ X & & K, X[K+1,n], Y & & X[1,K] \end{array}$$

The common goal of the parties is to write the sequence $X[K]Y$ on a shared output stream. Firstly, Alice is allowed to write, followed by Bob and then Charlie. The communication is one-way: Alice sends message M_A to Bob, and then Bob sends message M_B to Charlie.

From the presentation of the family of hard instances for FCNS decoding, it is easy to see that an algorithm for FCNS decoding that makes one pass over the input stream and one pass over the output stream can be used to obtain a communication protocol for INDEXCOPY. We state this as a fact.

FACT 7. *A streaming algorithm for FCNS decoding that makes one pass over the input stream and one pass over the output stream with space s serves as a communication protocol for* INDEXCOPY *with communication cost* $\mathrm{O}(s)$.

We will prove now that a communication protocol for INDEXCOPY has communication cost $\Omega(N)$.

LEMMA 6.1. *Every possibly randomized communication protocol for* INDEXCOPY *with error* $\mathrm{O}(1/N)$ *has communication cost* $\Omega(N)$.

PROOF. Let P be a (possibly randomized) communication protocol such that the parties output $X[K]Y$ on the shared output stream with error $\epsilon = \frac{1}{32n}$ on any input. We will prove now that P has communication cost $\Omega(n)$.

By Yao's minimax principle, there is a deterministic communication protocol P_d with distributional error at most ϵ that has the same communication complexity as P. Suppose that Alice's message in P_d is at most of length $n/100$ bits. We will show that under this assumption, for a particular input, Bob has to send a message of length $\Omega(n)$ bits which proves the theorem.

Since P_d has distributional error $\epsilon = \frac{1}{32n}$, we obtain by the Markov inequality:

$$\Pr_{x \leftarrow X}[\text{error} \geq 1/(16n) \mid X = x] \leq \frac{\epsilon}{1/(16n)} = \frac{1}{2}.$$

Therefore, there are at least $(1/2)2^n = 2^{n-1}$ values x for which the protocol errs with probability at most $1/(16n)$. Denote this set of x values by U. Furthermore, again by the Markov inequality:

$$\forall x \in U : \Pr_{k \leftarrow K}[\text{error} \geq 1/4 \,|\, X = x, K = k] \leq \frac{\frac{1}{16n}}{1/4} = \frac{1}{4n}.$$

Therefore, for any $x \in U$ and any $k \in [n]$, the protocol errs with probability less than $1/4$.

Consider an input of Alice x coming from U that is different from $x^0 = 0 \ldots 0$ and $x^1 = 1 \ldots 1$. Then Alice cannot write the bit $X[K]$ on the output stream. If on input x Alice writes deterministically 0 (or 1) then there is a value of K such that Alice wrote the wrong bit. The error on x would then be greater than $1/(16n)$ contradicting the fact that $x \in U$. Therefore, for all $x \in U \setminus \{x^0, x^1\}$ it is either Bob or Charlie who writes the bit $X[K]$.

We will argue now that there is at least one $x \in U$ and $k \in [n]$ such that Charlie has to output the bit $X[K]$.

Since the maximal message length of Alice is $n/100$ bits, there is a subset $U' \subseteq U$ with $|U'| \geq \frac{|U|}{2^{n/100}} = 2^{n-1-n/100}$ such that for all $x \in U'$, the message M_A sent by Alice is the same. Denote this message by m_A.

Using a technique of [Magniez et al. 2010], we will show now that there are $x_1, x_2 \in U'$ and $k \in [n]$ such that $x_1[k] \neq x_2[k]$ and $x_1[k+1, n] = x_2[k+1, n]$. This can be seen by building the following two-ranked tree: For each $x \in U'$ the tree has exactly one leaf at depth n that is labeled by x. All edges of the tree are labeled by bits 0 or 1. The sequence of labels of the edges of a path from a leaf to the root then equals the label of the leaf. An example for such a tree is provided in Figure 21.

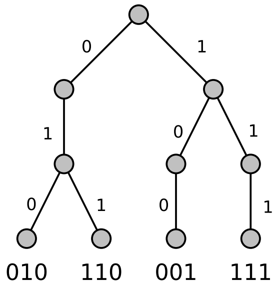


Fig. 21. Organizing the set $\{010, 110, 001, 111\}$ in a two-ranked tree.

By construction of the tree, the labels of leaves that have a common ancestor at depth i have the same suffix of length i. Consider now an inner node v of this tree with two children nodes. Such an inner node exists since the two-ranked tree has depth n and contains $|U'| \geq 2^{n-1-n/100}$ leaves. Let k be its depth. Furthermore, let x_1 be the label of an arbitrary leaf connected to the left child of v, and let x_2 be the label of an arbitrary leaf connected to the right child of v. Then $x_1[k+1, n] = x_2[k+1, n]$ and $x_1[k] \neq x_2[k]$ as desired.

Since $x_1, x_2 \in U'$, the protocols errs with probability at most $1/4$ if $X \in \{x_1, x_2\}$ and $K = k$. Note that on both inputs x_1 and x_2, Bob has the same suffix $X[K+1, n]$ since $x_1[k+1, n] = x_2[k+1, n]$. Furthermore, Bob receives the same message m_A of Alice. Hence, Bob can not distinguish between the two events $X = x_1$ and $X = x_2$ if $K = k$.

Since $x_1[k] \neq x_2[k]$, Bob can not output the bit $X[K]$ since this would lead to an error larger than $1/4$.

Therefore, in this setting Charlie has to output the bit $X[K]$. This also requires that subsequently Charlie outputs Y. Since Y is independent of the conditioning $X = x_1$ (or x_2) and $K = k$ and Charlie has no information about Y, we deduce that Charlie can learn Y from Bob's message M_B with error at most $3/4$.

Fix the input distribution $X \in \{x_1, x_2\}$, $K = k$ and Y is chosen uniformly at random. Then

$$\mathrm{H}(M_B) \geq \mathrm{H}(M_B) - \mathrm{H}(M_B \,|\, Y) = \mathrm{I}(M_B : Y),$$

where H is the entropy function and $\mathrm{I}(M_B : Y)$ denotes the mutual information between M_B and Y. Furthermore,

$$\mathrm{I}(M_B : Y) = \mathrm{H}(Y) - \mathrm{H}(Y \,|\, M_B) = n - \mathrm{H}(Y \,|\, M_B).$$

By the Fano Inequality, we obtain

$$\mathrm{H}(Y \,|\, M_B) \leq 1 + 1/4n.$$

This implies that $\mathrm{I}(M_B : Y) \geq 3/4n - 1$ and $\mathrm{H}(M_B) \in \Omega(n)$ which in turn implies that the average message length is $\Omega(n)$. □

Finally, we state our space lower bound for streaming algorithm for FCNS decoding that make one pass over the input stream and one pass over the output stream.

THEOREM 6.2. *Every randomized streaming algorithm for FCNS decoding that makes one pass over the input stream and one pass over the output stream with error probability* $\mathrm{O}(1/N)$ *requires space* $\Omega(N)$.

PROOF. The proof is by contradiction. Suppose that there is such a streaming algorithm with space $o(N)$. Then, by Fact 7 there is a communication protocol for INDEX-COPY with communication cost $o(N)$. This, however, is a contradiction to Lemma 6.1 that states that such a communication protocol has communication cost at least $\Omega(N)$. □

APPENDIX

A. ONE-PASS ALGORITHM WITH SPACE LINEAR IN THE DEPTH OF THE DOCUMENT

We discuss now a straight-forward one-pass streaming algorithm, Algorithm 10, that uses $\mathrm{O}(d)$ space to validate a well-formed XML document of depth d. Let (Σ, e, s_e) denote the input DTD, and for all $c \in \Sigma$ let A_c be a deterministic finite automaton with initial state q_e^0 and transition function δ_e that accepts words ω iff $e(c)$ accepts ω. Note that since we assume in this work that the size of the input DTD is $\mathrm{O}(1)$, the size of $(A_e)_{e\in\Sigma}$ and the time complexity to compute it is $\mathrm{O}(1)$.

In order to validate the input stream, we check for all nodes v that the sequence of labels of its children fulfills the regular expression $e(v)$ (remember: we write ambiguously v to denote the node as well as the label of v). We do this by feeding the sequence of labels of its children into automaton A_v, and we reject if the automaton does not accept this sequence.

Consider an internal node v at depth $\mathrm{depth}(v)$ with children $c_1, \ldots, c_k$. Then $vc_1 \ldots \overline{c_1}c_2 \ldots \overline{c_{k-1}}c_k \ldots \overline{c_k}\overline{v}$ is a substring of the input stream. As soon as we encounter the opening tag v, we store the initial state of A_v in an array S at index $\mathrm{depth}(v)$. As soon as an opening tag of a children node of v is encountered, we compute the follow-up state of $S[\mathrm{depth}(v)]$ by feeding the children node's label into A_v on state $S[\mathrm{depth}(v)]$. When $\overline{v}$ is reached and $S[\mathrm{depth}(v)]$ is not an accepting state of A_v, we report invalidity.

Since the depth of the document is d, there are at most d nodes whose validity has to be checked at the same time. These nodes are the nodes on the path from the current node to the root node. Therefore, we need to store at most d states of the automata $(A_\sigma)_{\sigma\in\Sigma}$ leading to a space complexity $\mathrm{O}(d)$. See Algorithm 10 for details.

Algorithm 10 One-pass Streaming Algorithm for VALIDITY with space $\mathrm{O}(d)$

```
Require: input stream is a well-formed XML document of depth $d$
$l \leftarrow -1$, $L, S \leftarrow$ array of size $d$
while stream not empty do
    $x \leftarrow$ next tag on stream
    if $x$ is an opening tag $c$ then
        if $l = -1$ then {Root node}
            if $c \neq s_e$ then report error and abort end if
        else {Node different from the root node}
            $S[l] \leftarrow \delta_{L[l]}(S[l], c)$
        end if
        $l \leftarrow l + 1$
        $L[l] \leftarrow c$
        $S[l] \leftarrow q_c^0$
    else {$x$ is a closing tag $\overline{c}$}
        if $l \neq -1$ then
            if $S[l]$ is not an accepting state of $A_c$ then report error and abort end if
        end if
        $l \leftarrow l - 1$
    end if
end while
```

THEOREM A.1. *Algorithm 10 is a deterministic one-pass streaming algorithm for* VALIDITY *with space* $\mathrm{O}(d)$ *and* $\mathrm{O}(1)$ *processing time per letter where* d *is the depth of the input XML document.*

PROOF. Correctness of the algorithm follows by construction. Concerning space, the arrays L and S are of size $\mathrm{O}(d)$ and since l does not exceed d, the space requirements for storing l are $\mathrm{O}(\log d)$. □

B. MISSING PROOFS OF SECTION 5

B.1. Proof of Lemma 5.1

PROOF. Recall Definition 2.3 of XML and Definition 2.4 of $\mathrm{XML}^{\mathrm{F}}$. We will show that the following two functions XML' and $\mathrm{XML}^{\mathrm{F}'}$ which, applied to the root of a tree t, generate the sequences of opening tags of $\mathrm{XML}(t)$ and $\mathrm{XML}(\mathrm{FCNS}(t))$ (without left/right annotations) are equivalent. For a tree t and nodes $x, x_1, \ldots, x_n$ we define

$$\begin{aligned}\mathrm{XML}'(x) &= x\,\mathrm{XML}'(\mathrm{children}(x)),\\ \mathrm{XML}'(x_1,\ldots,x_n) &= \mathrm{XML}'(x_1)\ldots\mathrm{XML}'(x_n),\\ \mathrm{XML}'(\perp) &= \epsilon,\end{aligned}$$

and

$$\begin{aligned}\mathrm{XML}^{\mathrm{F}'}(x) &= x\,\mathrm{XML}^{\mathrm{F}'}(\mathrm{fc}(x))\,\mathrm{XML}^{\mathrm{F}'}(\mathrm{ns}(x)),\\ \mathrm{XML}^{\mathrm{F}'}(\perp) &= \epsilon.\end{aligned}$$

Clearly, $\mathrm{XML}'(\mathrm{root}(t))$ and $\mathrm{XML}^{\mathrm{F}'}(\mathrm{root}(t))$ construct the sequences of opening tags of $\mathrm{XML}(t)$ and $\mathrm{XML}(\mathrm{FCNS}(t))$. Let $x \in t$ be any node. We prove the following statement by induction on the size of the subtree below x:

$$\mathrm{XML}'(x) = x\,\mathrm{XML}^{\mathrm{F}'}(\mathrm{fc}(x)). \tag{6}$$

The statement is trivially true if x is a leaf, that is a tree of size 1. Let x be a non-leaf node with children $x_1, \ldots, x_n$. Then

$$\begin{aligned}x\mathrm{XML}^{\mathrm{F}'}(\mathrm{fc}(x)) &= x\,x_1\,\mathrm{XML}^{\mathrm{F}'}(\mathrm{fc}(x_1))\,\mathrm{XML}^{\mathrm{F}'}(\mathrm{ns}(x_1)) &(7)\\ &= x\,\mathrm{XML}'(x_1)\,\mathrm{XML}^{\mathrm{F}'}(x_2) &(8)\\ &= x\,\mathrm{XML}'(x_1)\,x_2\,\mathrm{XML}^{\mathrm{F}'}(\mathrm{fc}(x_2))\,\mathrm{XML}^{\mathrm{F}'}(ns(x_2)) &(9)\\ &= x\,\mathrm{XML}'(x_1)\,\mathrm{XML}'(x_2)\,\mathrm{XML}^{\mathrm{F}'}(x_3) &(10)\\ &\ldots\\ &= x\,\mathrm{XML}'(x_1)\,\ldots\,\mathrm{XML}'(x_n) = x\,\mathrm{XML}'(\mathrm{children}(x)) = \mathrm{XML}'(x).\end{aligned}$$

We used the induction hypothesis in Equation 7 to obtain Equation 8 and in Equation 9 to Equation 10. Let r denote the root of t. Then using Equation 6 the result follows

$$\begin{aligned}\mathrm{XML}^{\mathrm{F}'}(r) &= r\mathrm{XML}^{\mathrm{F}'}(\mathrm{fc}(r))\mathrm{XML}^{\mathrm{F}'}(\mathrm{ns}(r))\\ &= \mathrm{XML}'(r)\mathrm{XML}^{\mathrm{F}'}(\perp) = \mathrm{XML}'(r).\end{aligned}$$

□

B.2. Proof of Lemma 5.13

PROOF. We will prove that for a node $x \in t$ the following is true

$$\mathrm{XML}(x) = x\,\mathrm{dec}(\mathrm{fc}(x))\,\overline{x}. \tag{11}$$

The proof is by induction on the height of the subtree below x and is similar to the proof of Lemma 5.1. The claim is obvious for leaves. Let x be a node and let $v_1, \ldots, v_n$ denote the children of x. Then

$$\begin{aligned} x\,\mathrm{dec}(v_1)\,\overline{x} &= x\,v_1\,\mathrm{dec}(\mathrm{fc}(v_1))\,\overline{v_1}\,\mathrm{dec}(v_2)\,\overline{x} && (12)\\ &= x\mathrm{XML}(v_1)\,v_2\,\mathrm{dec}(\mathrm{fc}(v_2))\,\overline{v_2}\,\mathrm{dec}(v_3)\,\overline{x} && (13)\\ &= x\mathrm{XML}(v_1)\,\mathrm{XML}(v_2)\,v_3\,\mathrm{dec}(\mathrm{fc}(v_3))\,\overline{v_3}\,\mathrm{dec}(v_4)\,\overline{x} && (14)\\ &\quad \ldots\\ &= x\,\mathrm{XML}(\mathrm{children}(x))\,\overline{x} = \mathrm{XML}(x), \end{aligned}$$

where we used the induction hypothesis in Equation 12 to obtain Equation 13, and in Equation 13 to obtain Equation 14. Since the root node r of the tree t does not have a next sibling, the result follows using Equation 11

$$\mathrm{dec}(r) \;=\; r\,\mathrm{dec}(\mathrm{fc}(r))\,\overline{r}\,\mathrm{dec}(\mathrm{ns}(r)) = \mathrm{XML}(r).$$

□

ACKNOWLEDGMENT

The authors would like to thank Michel de Rougemont, who, among other things, introduced the authors to the problem of validating streaming XML documents. Furthermore, the authors would like to thank the anonymous reviewers whose comments allowed us to substatially improve the quality of this article.

REFERENCES

ALON, N., MATIAS, Y., AND SZEGEDY, M. 1999. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences 58,* 1, 137–147.

BALMIN, A., PAPAKONSTANTINOU, Y., AND VIANU, V. 2004. Incremental validation of xml documents. *ACM Trans. Database Syst. 29,* 4, 710–751.

BAR-YOSSEF, Z., JAYRAM, T. S., KUMAR, R., AND SIVAKUMAR, D. 2004. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences 68,* 4, 702–732.

BARBOSA, D., MENDELZON, A. O., LIBKIN, L., MIGNET, L., AND ARENAS, M. 2004. Efficient incremental validation of xml documents. In *Proceedings of the 20th International Conference on Data Engineering*. ICDE '04. IEEE Computer Society, Washington, DC, USA, 671–.

BEAME, P. AND HUYNH-NGOC, D.-T. 2008. On the value of multiple read/write streams for approximating frequency moments. In *FOCS*. 499–508.

BEAME, P., JAYRAM, T., AND RUDRA, A. 2007. Lower bounds for randomized read/write stream algorithms. In *STOC*. 689–698.

DEMETRESCU, C., FINOCCHI, I., AND RIBICHINI, A. 2006. Trading off space for passes in graph streaming problems. In *ACM-SIAM SODA*. 714–723.

GROHE, M., HERNICH, A., AND SCHWEIKARDT, N. 2006. Randomized computations on large data sets: Tight lower bounds. In *ACM PODS*. 243–252.

GROHE, M., HERNICH, A., AND SCHWEIKARDT, N. 2009. Lower bounds for processing data with few random accesses to external memory. *Journal of the ACM 56,* 3, 1–16.

GROHE, M., KOCH, C., AND SCHWEIKARDT, N. 2005. The complexity of querying external memory and streaming data. In *FCT*. 1–16.

GROHE, M., KOCH, C., AND SCHWEIKARDT, N. 2007. Tight lower bounds for query processing on streaming and external memory data. *Theor. Comput. Sci. 380*, 199–217.

GROHE, M. AND SCHWEIKARDT, N. 2005. Lower bounds for sorting with few random accesses to external memory. In *ACM PODS*. 238–249.

KONRAD, C. AND MAGNIEZ, F. 2012. Validating XML documents in the streaming model with external memory. In *ICDT*. 34–45. Best Newcomer Paper.

KUSHILEVITZ, E. AND NISAN, N. 1997. *Communication complexity*. Cambridge University Press.

MAGNIEZ, F., MATHIEU, C., AND NAYAK, A. 2010. Recognizing well-parenthesized expressions in the streaming model. In *ACM STOC*. 261–270.

MARTENS, W., NEVEN, F., SCHWENTICK, T., AND BEX, G. J. 2006. Expressiveness and complexity of xml schema. *ACM Trans. Database Syst. 31,* 3, 770–813.

MUTHUKRISHNAN, S. 2005. *Data Streams: Algorithms and Applications*. Now Publishers Inc.

NEVEN, F. 2002. Automata theory for xml researchers. *Sigmod Record 31*, 2002.

PAPAKONSTANTINOU, Y. AND VIANU, V. 2000. DTD inference for views of XML data. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. PODS '00. ACM, New York, NY, USA, 35–46.

SEGOUFIN, L. AND SIRANGELO, C. 2007. Constant-memory validation of streaming XML documents against DTDs. In *ICDT*. 299–313.

SEGOUFIN, L. AND VIANU, V. 2002. Validating streaming XML documents. In *ACM PODS*. 53–64.