Validating XML Documents in the Streaming Model with External Memory*

Christian Konrad
LIAFA, Univ. Paris Diderot; Paris, France;
and Univ. Paris-Sud; Orsay, France.
konrad@lri.fr

Frédéric Magniez
LIAFA, Univ. Paris Diderot, CNRS; Paris, France.
magniez@univ-paris-diderot.fr

ABSTRACT

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 $O(\log^2 N)$, a constant number of auxiliary read/write streams, and $O(\log N)$ 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 in the usual streaming model without external memory can be done with sublinear memory. There is a one-pass algorithm using $O(\sqrt{N \log N})$ space, and a bidirectional two-pass algorithm using $O(\log^2 N)$ space performing 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

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 [1], 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 [8, 10, 6]. Then the streaming algorithm is also relaxed 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.

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 $O(\log N)$-pass deterministic streaming algorithms with constant memory space and a constant number of auxiliary streams [7]. 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 [7] and then extended to two-sided error [4, 3].

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 few is known on 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 [7, 4, 3], meaning that $\Omega(\log N)$ passes are necessary. For the upper bound, only simple refinements of the direct algorithm are known: no auxiliary stream, one pass and linear memory in the height of the XML document, which in the worst case is as large as N.

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. Prior

*Supported by the French ANR SeSur and Defis programs under contracts ANR-07-SESU-013 (VERAP project) and ANR-08-EMER-012 (QRAC project). Christian Konrad is supported by a Fondation CFM-JP Aguilar grant.

ICDT 2012, March 26–30, 2012, Berlin, Germany.

works on that topic [16, 15] essentially try to characterize those DTDs for which validity can be checked by a finite-state automaton, that is a one-pass deterministic streaming algorithm with constant memory. Concerning arbitrary DTDs, two approaches have been considered in [16]. The first one leads to an algorithm with memory space linear in the height of the XML document [16]. The second one consists in 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 later requires that both, documents and DTD, are converted before validation.

One of the obstacles prior works had to cope with was to check well-formedness of XML documents, that is every opening tag matches its same-level closing tag. Due to the past work of [12], we can now perform such a verification with a constant-pass randomized streaming algorithm with sublinear memory space and no auxiliary streams. In one-pass the memory space is $O(\sqrt{N \log N})$, and collapses to $O(\log^2 N)$ with an additional pass in reverse direction.

The starting point of this work is the fact that checking validity is hard without auxiliary streams. There are DTDs defining ternary XML documents for which any p-pass bidirectional randomized streaming algorithm requires $\Omega(N/p)$ space. 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 [9], a similar approach using ternary trees with a reduction to Set-Disjointness is used for proving lower bounds for queries. For the sake of completeness we provide a proof in Appendix A.

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. For the case of XML documents encoding binary trees, we present in Section 3 two 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 show how to design a one-pass algorithm with space $O(\sqrt{N \log N})$ (**Theorem 1**). We conjecture that there is a $\Omega(N^\alpha)$ lower bound for one-pass algorithms for some $\frac{1}{3} \leq \alpha \leq \frac{1}{2}$. With a second pass in reverse direction the memory collapses to $O(\log^2 N)$ (**Theorem 2**). These two algorithms make use of the simple, but fundamental fact that in one pass over an XML document each node is seen twice in form 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 by their closing tags. Our algorithms exploit this observation.

Then, in Section 4 we present our main result. **Corollary 1** states that validity 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 $O(\log N)$ passes over these streams. Validity of a node depends on its children, hence is crucial to have easy access to the sequence of children of any node. The fundamental idea to establish this is, firstly, to compute the First-Child-Next-Sibling (FCNS) encoding, an encoding as a 2-ranked tree of the XML document. 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 3**). Since the FCNS encoding can be seen as a reordering of the tags of the original document, our strategy is to see this problem as a sorting problem with a particular comparison function. Merge sort can be implemented as a streaming algorithm, and we make use of it customized by an adapted merge function. The same idea can be used for decoding with similar complexity (**Theorem 7**).

Then, based on the FCNS encoding, verification can be done either in one pass and $O(\sqrt{N \log N})$ space (**Theorem 4**), or in two bidirectional passes and $O(\log^2 N)$ space (**Theorem 5**). Concerning FCNS encoding and decoding, we show a linear space lower bound for one-pass algorithms. For decoding, we present a $O(\sqrt{N \log N})$ algorithm (**Theorem 6**) that performs one pass over the input, but two passes over the output. We conjecture for encoding that memory space remains $\Omega(N)$ after any constant number of passes, which 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 in sublinear space. For user interactions, the original document can be obtained by the sublinear space 3-pass algorithm. The applicability of this idea is left as an open question.

2. PRELIMINARIES

From now Σ is 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 $1 \leq i_1 < i_2 < ... < i_k \leq N$.

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, as the case may be, to read-write external memory as in [7, 5]. See also the review in [8]. We assume that any letter of Σ fits into one cell of internal/external memory. The external memory is a collection of auxiliary streams, that we see as *read/write streams* with, again, sequential access. When needed, we augment the alphabet of auxiliary streams from Σ by k-tuples of elements in $\Sigma \cup [0, 2N]$, for some fixed constant k, which therefore fit in one cell of auxiliary streams.

At the beginning of each pass on a read/write stream, the algorithm decides whether it performs 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, opposed to unidirectional streaming algorithm where each pass is in the same order, the algorithm can decide the direction of the sequential access.

For 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 [13] for an introduction to streaming algorithms.

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

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

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

2.2 XML documents

We consider finite unranked ordered labeled trees t, where each tree node is labeled by some label in Σ, and its root has a distinguished label r. Moreover, the children of every non-leaf node are ordered. From now, we omit the terms ordered and labeled. Then k-ranked trees are a special case where each node has at most k children. Binary trees are a special type of 2-ranked trees, where each node is either a leaf or has exactly 2 children.

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 traversal of t (Figure 1): 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 XML(t).

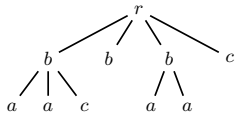

Figure 1: Let $\Sigma = \{a, b, c\}$, and let t be the tree as above. Then $\text{XML}(t) = rba\overline{a}a\overline{a}c\overline{c}\overline{b}b\overline{b}ba\overline{a}a\overline{a}\overline{b}c\overline{c}\overline{r}$.

We assume that the input XML sequences X are *well-formed*, namely $X = \text{XML}(t)$, for some tree t. The past work of [12] 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 we will show that randomness is useless for validation.

Let us introduce more useful notations. 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 $X = \text{XML}(t)$ corresponds to a unique tree node v of t. We sometimes denote v either by $X[i]$ or $X[j]$. We also write (ambiguously) v for its corresponding opening tag, and $\overline{v}$ for its corresponding closing tag. Then, the *position* of v in X is $\text{pos}(v) = i$. Similarly, $\text{pos}(\overline{v}) = j$.

We consider XML validity against some DTD. A DTD is a mapping D from Σ to regular expressions over Σ. Let t be a tree. Then a node $v \in t$ with label v and children $v_1, v_2, \ldots, v_k$ with respective labels $v_1, v_2, \ldots, v_k$ is *valid against D* if $v_1, v_2, \ldots, v_k$ satisfies the regular expression $D(v)$. In particular, v can be a leaf if and only if the empty word ε satisfies the regular expression $D(v)$. Then t is *valid against D* if all its nodes are valid against D. 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 (VALIDITY). *Let D be some DTD. The problem* VALIDITY *consists of deciding whether an input tree t given by its XML sequence* 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. 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 Algorithm 1 and the two-pass Algorithm 2 (indeed in the subroutine Algorithm 3), that we do not describe here.

We fix now a DTD D, and assume that, in our algorithms, we have access to a procedure check(a, b, c) that signalizes invalidity and aborts if bc is not valid against the regular expression $D(a)$. Otherwise it returns without any action.

3.1 One-pass algorithm

In order to validate an XML document, we ensure validity of all tree nodes. For checking validity of a node v with two children, we have to relate 3 labels, that is the label v of the node itself, and the labels of the two children nodes v_1, v_2. 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. Algorithm 1 makes use of the fact that there are these two chances to verify a node. 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 requirements depend 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 1 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 $O(\log N)$ space. Opening tags don't require the storage of their depth (we store a depth of -1 which we assume to require only constant space).

Algorithm 1 Validity of binary trees in 1-pass

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, ϵ, ϵ) **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** $|\{(a, -1) \in S \mid a \text{ opening }\}| \geq K$ **then**
11: remove bottom-most $(a, -1)$ in S, a opening
12: **end if**
13: $d \leftarrow d + 1$
14: push $(x, -1)$
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** pop S
21: **end if**
22: **if** S has on top $(c, -1)$ **then** pop S **end if**
23: push (x, d)
24: **end if**
25: **end while**

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 2 visualizes the different cases with their stack modifications appearing in Algorithm 1.

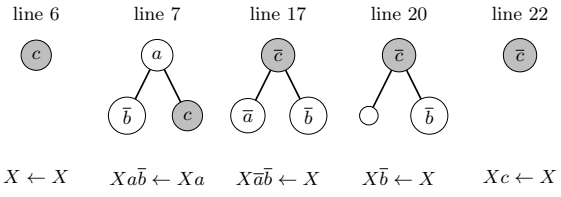

Figure 2: Visualization of the different conditions in Algorithm 1 with the applied stack modifications. X represents the bottom part of the stack. Note that Algorithm 1 pushes the currently treated tag c or $\overline{c}$ on the stack in Line 14 or Line 23. c or $\overline{c}$ corresponds to the highlighted node.

Fact 1 (not proved here) and Lemma 1 concern the structure of the stack S used in Algorithm 1.

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. $\text{pos}(x_1) < \text{pos}(x_2) \cdots < \text{pos}(x_k)$,
2. $\text{depth}(x_1) \leq \text{depth}(x_2) \cdots \leq \text{depth}(x_k) \leq d$. *Moreover, if* $\text{depth}(x_i) = \text{depth}(x_{i+1})$ *then x_i is the left sibling of x_{i+1},*
3. *The sequence $x_1 \dots x_k$ satisfies the regular expression $\overline{a}^* b^* (\epsilon \mid \overline{c} \mid \overline{d}\overline{e})$, 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 only removed from S upon verification of its parent node.*

LEMMA 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. 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 one. Then* $\text{pos}(\overline{c_{i+1}}) \geq \text{pos}(\overline{c_i}) + 2K$.

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 $X[\text{pos}(\overline{c_{i+1}}) + 1]$ 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 $\text{pos}(\overline{c_i}) < k < \text{pos}(\overline{c_{i+1}})$. 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 $\text{pos}(\overline{c_{i+1}}) \geq \text{pos}(\overline{c_i}) + 2K$. □

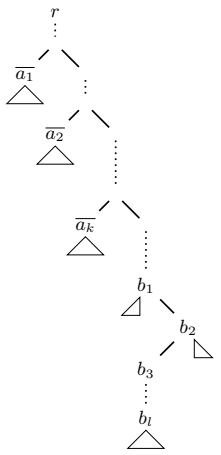

Figure 3: Visualization of the structure of the stack used in Algorithm 1. The stack fulfills the regular expression $\overline{a}^* b^* (\epsilon \mid \overline{c} \mid \overline{d}\overline{e})$, compare Item 3 of Fact 1. The $(\overline{a_i})_{i=1\dots k}$ are closing tags whose parents' nodes were not verified top-down. For $j > i$, a_j is connected to a_i by the right sibling of a_i. The $(b_i)_{i=1\dots l}$ 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 1 provide more insight in the stack structure and are used in the proof of Theorem 1. 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 1 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 1. *Algorithm 1 is a one-pass streaming algorithm for* VALIDITY(2) *with space* $O(\sqrt{N \log N})$ *and* $O(1)$ *processing time per letter.*

PROOF. To prove correctness, we have to ensure validity of all nodes. Leaves are 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 23, 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 1 states that the stack contains at most one right closing tag. From Item 4 of Fact 1 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 1 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 1. 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.

3.2 Two-pass algorithm

The bidirectional two-pass Algorithm 2 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 $\overline{c_1}, c_2$ appears in the stream, a 4-tuple consisting of $\overline{c_1}, c_2, \mathrm{depth}(c_1)$ and $\mathrm{pos}(\overline{c_1})$ gets 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, namely eventually in parallel.

Algorithm 2 Two-pass algorithm validating binary trees

run **Algorithm 3** reading the stream from left to right
run **Algorithm 3** reading the stream from right to left, where opening tags are interpreted as closing tags, and vice versa.

Algorithm 3 Validating nodes with size(left subtree) ≥ size(right subtree)

```
1: l ← 0; n ← 0; S ← empty stack
2: while stream not empty do
3:    x ← next tag on stream (and move stream to next tag)
4:    y ← next tag on stream, without consuming it yet
5:    n ← n + 1
6:    if x is an opening tag c then
7:       l ← l + 1
8:       if y = c̄ then check(c, ε, ε) end if
9:    else {x is a closing tag c̄}
10:      l ← l − 1
11:      if S has on top (·, ·, l + 1, ·) then
12:         (ā, b, ·, ·) ← pop from S; check(c, a, b)
13:      end if
14:      if y is an opening tag d then
15:         push (c̄, d, l, n) to S
16:      end if
17:   end if
18:   while there is s1 = (·, ·, ·, n1) just below s2 = (·, ·, ·, n2) in
      S with n − n2 > n2 − n1 do
19:      suppress s2 from S
20:   end while
21: end while
```

Figure 4 visualizes the different cases in Algorithm 3.

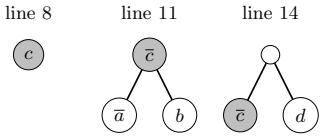

Figure 4: Visualization of the different conditions in Algorithm 3. The incoming tag x corresponds to the highlighted node.

We highlight some properties concerning the stack used in Algorithm 3.

FACT 2. *S in Algorithm 3 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 5 illustrates the relationship between two consecutive stack elements as discussed in Item 1 of Fact 2.

LEMMA 2. *Algorithm 3 verifies all nodes v 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

$$\mathrm{pos}(\overline{a_1}) - \mathrm{pos}(a_1) \geq \mathrm{pos}(\overline{b_1}) - \mathrm{pos}(b_1), \qquad (1)$$

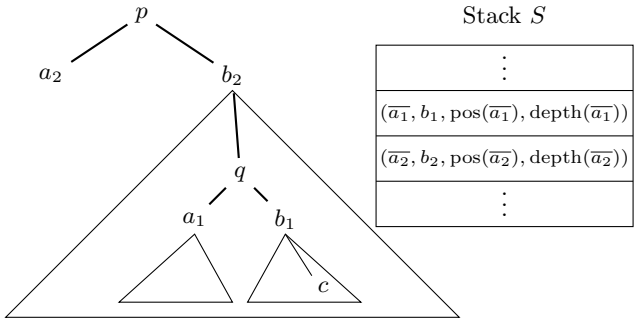

Figure 5: c is the current element under consideration in Algorithm 3. a_1, b_1 is in the subtree of b_2, compare Item 1 of Fact 2.

.

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 3 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 c be the current tag under consideration such that $\mathrm{pos}(b_1) < \mathrm{pos}(c) < \mathrm{pos}(\overline{q})$. The situation is visualized in Figure 5.

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

$$\mathrm{pos}(c) - \mathrm{pos}(\overline{a_1}) > \mathrm{pos}(\overline{a_1}) - \mathrm{pos}(\overline{a_2}). \qquad (2)$$

Note that the left side of Inequality 2 is a lower bound on the size of the right subtree of q. Furthermore, the right side of Inequality 2 upper bounds the size of the left subtree of q.

Using $\mathrm{pos}(c) - \mathrm{pos}(\overline{a_1}) \leq \mathrm{pos}(\overline{b_1}) - \mathrm{pos}(b_1) + 1$ and $\mathrm{pos}(\overline{a_1}) - \mathrm{pos}(\overline{a_2}) > \mathrm{pos}(\overline{a_1}) - \mathrm{pos}(a_1)$, Inequality 2 contradicts Inequality 1 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 2. *Algorithm 2 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 2, we ensure that all nodes get verified. By Lemma 2, 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 3, 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 3, 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 3 uses space $\mathrm{O}(\log^2 N)$. This also implies that the while-loop at line 18 of Algorithm 3 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. □

4. VALIDITY OF GENERAL TREES

4.1 Preparation

The *FCNS encoding* (see for instance [14]) 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, for respectively *left* and *right* opening/closing tags. 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.

Instead of annotating by left/right, another way to uniquely identify a node as left or right is to insert dummy leaves with label $\perp$. For a tree t, we denote the binary version without annotations and insertion of $\perp$ leaves by $\mathrm{FCNS}^{\perp}$. The two representations can be easily transformed into each other. In this section, we compute the FCNS encoding with annotations. In the next section, we present algorithms for the validation of the encoded form that make use of the representation using dummy leaves. See Figure 6 for an example.

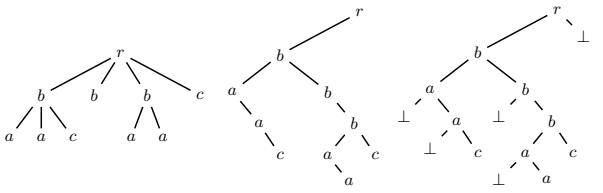

Figure 6: Left: introductory example tree t already shown in Figure 1. Middle: FCNS encoding of t: $\mathrm{XML}(\mathrm{FCNS}(t)) = r_L b_L a_L a_R c_R \overline{c_R} \overline{a_R} \overline{a_L} b_R b_R a_L a_R \overline{a_R a_L} c_R \overline{c_R} \overline{b_R} \overline{b_R} \overline{b_L} \overline{r_L}$. Right: $\mathrm{FCNS}^{\perp}$ encoding of t: $\mathrm{XML}(\mathrm{FCNS}^{\perp}(t)) = rba\perp\overline{\perp}a\perp\overline{\perp}c\overline{ca}\overline{a}b\perp\overline{\perp}ba\perp\overline{\perp}a\overline{a}\overline{a}c\overline{c}\overline{b}\overline{b}\overline{b}\perp\overline{\perp}\overline{r}$.

In the following subsections we provide streaming algorithms for the transformation of $\mathrm{XML}(t)$ to $\mathrm{XML}(\mathrm{FCNS}(t))$, that we call the *FCNS encoding*, and its inverse, the *FCNS decoding*.

The FCNS encoding can be seen as a reordering of the tags of $\mathrm{XML}(t)$ and an annotation of the tags with left/right.

We state several properties about the relationship of the ordering of the tags in XML(t) and XML(FCNS(t)). Fact 3 concerns the structure of the subsequence of opening tags in XML(FCNS(t)), Fact 4 concerns the structure of the subsequence of closing tags in XML(FCNS(t)), and Fact 5 concerns the interplay of the subsequences of opening and closing tags in XML(FCNS(t)).

FACT 3. *The opening tags in* XML(t) *are in the same order as the opening tags in* XML(FCNS(t)).

For a node v of some tree t, let $\mathrm{pos}'(v)$ and $\mathrm{pos}'(\overline{v})$ be the respective positions of the opening and closing tags of v in XML(FCNS(t)).

FACT 4. *Nodes v_1, v_2 of t satisfy $\mathrm{pos}'(\overline{v_1}) < \mathrm{pos}'(\overline{v_2})$ iff one of the following conditions holds:*

1. *v_1 is in the subtree of v_2 in t;*
2. *or v_1 is a right sibling of v_2 in t;*
3. *or there is a node u with $\mathrm{depth}(u) \leq \mathrm{depth}(v_1) - 2$ such that $\mathrm{pos}(v_1) < \mathrm{pos}(u) \leq \mathrm{pos}(v_2)$.*

FACT 5. *Nodes v_1, v_2 of t satisfy $\mathrm{pos}'(\overline{v_1}) < \mathrm{pos}'(v_2)$ iff there is a node u with $\mathrm{depth}(u) \leq \mathrm{depth}(v_1) - 2$ such that $\mathrm{pos}(v_1) < \mathrm{pos}(u) \leq \mathrm{pos}(v_2)$.*

4.2 FCNS encoding

In this section, we are interested in computing the transformation XML(t) $\rightarrow$ XML(FCNS(t)). Our strategy is to compute the subsequence of opening tags of XML(FCNS(t)) (using Fact 3 and discussed in subsection 4.2.1) and the subsequence of closing tags (using Fact 4 and discussed in 4.2.2) of XML(FCNS(t)) independently, and merge them afterwards (using Fact 5 and discussed in subsection 4.2.3).

4.2.1 Computing the sequence of opening tags

Concerning the opening tags, since due to Fact 3 the subsequences of opening tags in XML(t) and XML(FCNS(t)) coincide, we extract the subsequence of opening tag of XML(t), and we annotate them with left or right as they should be in XML(FCNS(t)). Remind 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 $\mathrm{pos}(c)$. We summarize this as a fact:

FACT 6. *There is a streaming algorithm with space* $O(\log N)$ *that, given* XML(t) *as input, outputs on an auxiliary stream the sequence of opening tags of* XML(FCNS(t)) *with left/right annotations, and furthermore, annotates each tag c with* $\mathrm{depth}(c)$ *and* $\mathrm{pos}(c)$*, performing one pass on each stream.*

4.2.2 Computing the sequence of closing tags

For computing the sequence of closing tags, we start with the sequence of opening tags of XML(FCNS(t)) as produced by the output of the algorithm of Fact 6, that is, correctly annotated with left/right and with depth and position annotations. To obtain the correct subsequence of closing tags as in XML(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 $O(\log(N))$ passes and 3 auxiliary streams [7]. For the sake of simplicity, Algorithm 4 assumes an input of length 2^l for some $l > 0$.

Algorithm 4 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** $j = 1 \ldots 2^{l-i-1}$ merge(2^i) **end for**
4: **end for**

merge(b) reads simultaneously the next b values from stream 2 and stream 3, and merges them onto stream 1. The whole loop in Line 3 of Algorithm 4 requires one read pass on stream 2, one read pass on stream 3, and one write pass on stream 1. See Figure 7 for an illustration.

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

Figure 7: In Line 2, blocks from stream 1 are copied onto stream 2 and stream 3. The B_i are sorted blocks. In line 3, 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 $\overline{c_1}, \overline{c_2}$, decides whether $\mathrm{pos}'(\overline{c_1}) < \mathrm{pos}'(\overline{c_2})$. Firstly, consider nodes v_1, v_2 with $\mathrm{pos}(v_1) < \mathrm{pos}(v_2)$ to be as in Point 1 or Point 2 of Fact 4, that is, either v_1, v_2 are siblings or one node is contained in the subtree of the other one. Obviously, their ordering with respect to pos′ can easily be decided by their depth: $\mathrm{pos}'(\overline{v_1}) < \mathrm{pos}'(\overline{v_2})$ iff $\mathrm{depth}(v_1) > \mathrm{depth}(v_2)$.

If neither v_1, v_2 are siblings, nor v_2 is in the subtree of v_1 (Point 3 of Fact 4), then $\mathrm{pos}'(\overline{v_1}) < \mathrm{pos}'(\overline{v_2})$, independently of their depths. A comparison function hence should be able to infer the relationship of the two nodes, however, this seems to be difficult in the streaming model.

To overcome this problem, instead of defining a comparison function, we design a complete merge function in Lemma 3 that, by construction, only compares two nodes of the first kind. The key idea is to introduce *separator* tags which we denote by a new tag outside Σ. They are initially inserted right after each closing tag of a last child u, that is exactly before the depth decreases. We denote by $\overline{\overline{u}}$ the separator we introduce when seeing the last child u, and we define $\mathrm{depth}(\overline{\overline{u}}) = \mathrm{depth}(u)$.

FACT 7. *There is a streaming algorithm with space* $O(\log N)$ *that, given a sequence* XML(t) *on a stream, computes on another stream the sequence of opening tags* XML(FCNS(t)) *together with their separators, and annotated with* depth, pos *and left/right, performing one pass on each 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:

$$\mathrm{pos}'(\overline{v}) < \mathrm{pos}'(\overline{\overline{v_k}}) \quad \text{iff} \quad \mathrm{pos}'(\overline{v}) \leq \mathrm{pos}'(\overline{v_1}); \tag{3}$$

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

$$\mathrm{pos}'(\overline{\overline{v_k}}) < \mathrm{pos}'(\overline{\overline{w_k}}) \quad \text{iff} \quad \mathrm{pos}'(v_k) < \mathrm{pos}'(w_k). \tag{4}$$

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 XML(FCNS(t)) appear in the same order, and for all $\overline{v_1}, \overline{v_2} \in B$ with $\text{pos}(v_1) < \text{pos}(v_2)$, all closing tags $\overline{v}$ of nodes v with $\text{pos}(v_1) < \text{pos}(v) < \text{pos}(v_2)$ 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 $\overline{v_2} \in B_2$ $\text{pos}(v_1) < \text{pos}(v_2)$ is satisfied, and furthermore, all closing tags $\overline{v}$ of nodes v with $\text{pos}(v_1) < \text{pos}(v) < \text{pos}(v_2)$ are either in B_1 or B_2.

The only function to design is a comparator deciding for two closing tags $\overline{v_1}, \overline{v_2}$ from a well-sorted adjacent pair (B_1, B_2) whether $\text{pos}'(\overline{v_1}) < \text{pos}'(\overline{v_2})$.

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

LEMMA 3. *Let (B_1, B_2) be a well-sorted adjacent pair, and let $v_1 = B_1[p_1]$ and $v_2 = B_2[p_2]$ for some p_1, p_2. Assume that $\text{pos}'(v) < \text{pos}'(v_1)$ and $\text{pos}'(v) < \text{pos}'(v_2)$, for all $v \in B_1[1, p_1 - 1] \cup B_2[1, p_2 - 1]$. Then:*

1. *If v_1 is a separator, or there is a separator in B_1 after v_1, then $\text{pos}'(v_1) < \text{pos}'(v_2)$;*
2. *Else if v_2 is a separator then:*
   (a) *if $\text{depth}(v_1) < \text{depth}(v_2)$ then $\text{pos}'(v_1) < \text{pos}'(v_2)$,*
   (b) *else $\text{pos}'(v_1) > \text{pos}'(v_2)$;*
3. *Else (neither v_1 nor v_2 is a separator):*
   (a) *if $\text{depth}(v_1) \leq \text{depth}(v_2)$ then $\text{pos}'(v_1) < \text{pos}'(v_2)$,*
   (b) *else $\text{pos}'(v_1) > \text{pos}'(v_2)$.*

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 $l = 0$. First, we prove Point 1. Since B_1 is well-ordered, we only need to check that $\text{pos}'(B_1[l]) < \text{pos}'(B_2[1])$. 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 (3) it suffices to show that $\text{pos}'(\overline{u'}) < \text{pos}'(B_2[1])$. Clearly, the shortest path from u' to $B_2[1]$ passes by a common ancestor p of u' and $B_2[1]$ which is not the parent of u' since the separator $B_1[l]$ indicates that the last child u has been seen. Then, by the third condition of Fact 4, we get $\text{pos}'(\overline{u'}) < \text{pos}'(B_2[1])$.

For proving Points 2 and 3 we use the observation that if the premises to Point 1 are not fulfilled, v_1, v_2 do not have a common ancestor p s.t. $\text{pos}(v_1) < \text{pos}(p) < \text{pos}(v_2)$ and p is not the parent node of v_1. Furthermore, this observation implies that $\text{depth}(v_2) \geq \text{depth}(v_1) - 1$ and hence, if $\text{depth}(v_2) > \text{depth}(v_1)$ then v_2 is in the subtree of v_1. This and Fact 4 prove Points 2a, 2b, 3a and 3b.

We prove the observation by contradiction. Assume that there is such a node p. Since (B_1, B_2) is a well-ordered adjacent pair and $\text{pos}(v_1) < \text{pos}(p) < \text{pos}(v_2)$, node p would be in $B_1 \cup B_2$. Therefore, the separator $\overline{\overline{u}}$ inserted after the rightmost sibling of v_1 would be also in $B_1 \cup B_2$ as well. More precisely, this separator would be in $B_2[1 \dots p_2 - 1]$ since otherwise Point 1 would have been applied. This, however, is a contradiction to the assumption that $\text{pos}'(v) < \text{pos}'(v_1)\, \forall v \in B_1[1 \dots p_1 - 1] \cup B_2[1 \dots p_2 - 1]$ since it holds that $\text{pos}'(v_1) < \text{pos}'(\overline{\overline{u}})$. Hence such a node does not exist. □

LEMMA 4. *There is a $O(\log N)$-pass streaming algorithm with space $O(\log N)$ 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 Fact 7, we compute on the first auxiliary stream the sequence of opening tags with corresponding annotations, together with separators, and interpret opening tags as closing tags.

We show that we can do a merge sort algorithm with a merge function inspired by Lemma 3 on the first three auxiliary streams with $O(\log N)$ space and passes. For that assume that the first stream contains a sequence $(B_1, B_2, \dots, B_M)$ 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 3. Note that $(B_{2k-1}, B_{2k})_i$ are all well-formed pairs. Let $l = \max\{i : B_{2k-1}[i] \text{ is a separator}\}$. 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 Point 1 of Lemma 3. Then, we merge blocks $B_{2k-1}[l+1, 2^i]$ and B_{2k} by using the comparison function defined in Points 2 and 3 of Lemma 3. □

4.2.3 Merging opening and closing tags

Merging the subsequence of opening tags of XML(FCNS(t)) and the subsequence of closing tags of XML(FCNS(t)) can be done using one additional pass.

LEMMA 5. *There is a streaming algorithm with space $O(\log N)$ that, given the sequence of opening tags of XML(FCNS(t)) on a stream, and the sequence of closing tags of XML(FCNS(t)) on another stream, computes XML(FCNS(t)) on a third stream using one pass on each stream.*

PROOF. We directly apply Fact 5, so that we know when to alternate from the sequence of opening tags to the one of closing tags, and conversely. □

From Fact 6, Lemma 4 and Lemma 5 we obtain Theorem 3.

THEOREM 3. *There is a $O(\log N)$-pass streaming algorithm with space $O(\log N)$ and 3 auxiliary streams and $O(1)$ 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 4 the sequence of closing tags and we store them on auxiliary stream 1. Then, by Fact 6 we extract the sequence of opening tags, and we store them on auxiliary stream 2. By Lemma 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 $O(\log N)$. The processing time per letter of these operations is constant. □

4.3 Checking Validity on the encoding form

In this section, we reuse the algorithms for validating binary trees for the validation of the encoded form. We discuss one-pass read/write streaming algorithms (one for left-to-right passes, and one for right-to-left passes) that read XML(FCNS$^\perp(t)$) and output an XML document with annotations on closing tags that can be fed into Algorithm 1 or Algorithm 2. This requires little modifications in Algorithm 1 and Algorithm 2 since validity then depends on the annotations. Since we want to reuse the algorithms for the validation of binary trees, we suppose that XML(FCNS$^\perp(t)$) is available as input. The algorithm stated in Theorem 3 can be easily adapted such that it outputs XML(FCNS$^\perp(t)$) instead of XML(FCNS(t)).

The problem of validating the encoded form and the problem of validating binary trees are similar. Note that the children $v_1, \dots v_k$ of a node v form a substring in XML(FCNS$^\perp(t)$), see Figure 8. Hence, for validating a node v, the label v has to be related to the *block* of children nodes $\overline{v_k} \dots \overline{v_1}$. This is similar to the task of the validation of binary trees where the parent label v has to be related to the block of children nodes $\overline{v_1} v_2$.

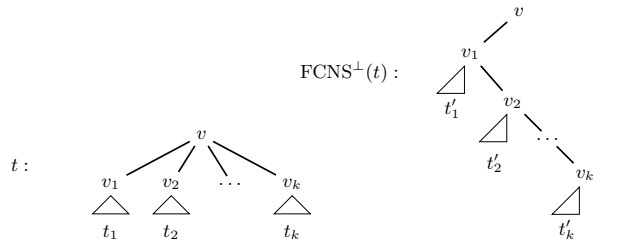

Figure 8: A tree t and its FCNS$^\perp$ encoding. While the opening and closing tags of the children of a node v are separated by the subtrees $t_1, \dots t_k$ in XML(t), the closing tags of the children of v are consecutive in XML(FCNS$^\perp(t)$) in reverse order, that is $\overline{v_k v_{k-1}} \dots \overline{v_2 v_1}$ is a substring of XML(FCNS$^\perp(t)$).

For a node v, we gather the information of the children nodes $v_1, \dots, v_k$ by the help of finite automata $\mathcal{A}_1$ (for left-to-right passes) and $\mathcal{A}_2$ (for right-to-left passes) that we define later, and the information - a state of the automata - is annotated at the closing tag of leaf v_1 (left-to-right) or v_k (right-to-left). Then, by the help of Algorithm 1 or Algorithm 2, this information is related to the parent label v.

We define automata $\mathcal{A}_1$ and $\mathcal{A}_2$ now. $\mathcal{A}_1$ is constructed from automaton A. Let $A = (\Sigma, Q, q_0, \delta, F)$ be 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. For $a \in \Sigma$ and the input DTD D, 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 9 to an automaton A that accepts words ω' such that $\omega' = a\omega$, $a \in \Sigma, \omega \in \Sigma^*$ if $\omega \in D(a)$.

Let $\mathcal{A}_1 = (\Sigma, Q_1, (q_0)_1, \delta_1, F_1)$ be a deterministic finite automaton that accepts a word ω, iff ω^{rev} is accepted by A, where ω^{rev} denotes ω read from right to left.

Let $\mathcal{A}_2 = (\Sigma, Q_2, (q_0)_2, \delta_2, F_2)$ be a deterministic finite automaton that accepts a word ω' such that $\omega' = \omega a$, $a \in \Sigma, \omega \in \Sigma^*$ if $\omega \in D(a)$. $\mathcal{A}_2$ is a version of the automaton in the right illustration of Figure 9 without ϵ transitions.

In the following, we assume for every state $q_1 \in Q_1$ and $q_2 \in Q_2$ that $\delta_1(q_1, \perp) = q_1$ and $\delta_2(q_2, \perp) = q_2$.

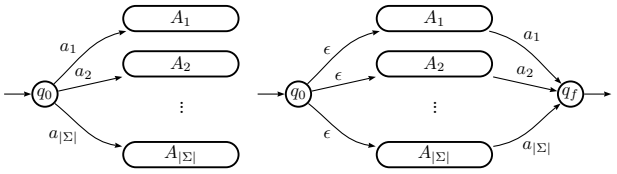

Figure 9: Left: Automaton A. $\mathcal{A}_1$ accepts words ω if A accepts ω^{rev}. Right: Automaton $\mathcal{A}_2$ is a version of the illustrated automaton without ϵ transitions.

Given XML(FCNS$^\perp(t)$), for a left-to-right pass, we annotate closing tags $\overline{v}$ by a state from the state set Q_1 of automaton $\mathcal{A}_1$. We denote the annotation for left-to-right passes of $\overline{v}$ by $\text{ann}_1(\overline{v})$.

if v is a leaf then $\text{ann}_1(\overline{v}) = \delta_1((q_0)_1, v)$,
otherwise let u be the right child of v, then $\text{ann}_1(\overline{v}) = \delta_1(\text{ann}_1(\overline{u}), v)$.

For a right-to-left pass, we annotate closing tags $\overline{v}$ by a state from the state set Q_2 of automaton $\mathcal{A}_2$. We denote the annotation of right-to-left passes of $\overline{v}$ by $\text{ann}_2(\overline{v})$.

if v is a left child then $\text{ann}_2(\overline{v}) = \delta_2((q_0)_2, v)$,
if v is a right child of u then $\text{ann}_2(\overline{v}) = \delta_2(\text{ann}_2(\overline{u}), v)$.

For the sake of completeness, the root can be annotated by $\text{ann}_2(\overline{r}) = (q_0)_2$, though this annotation will not be used for checking validity. Figure 10 shows the annotations of the children nodes $v_1, \dots, v_k$ of node v.

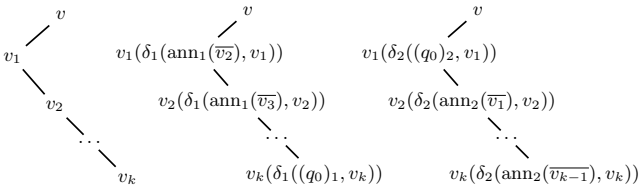

Figure 10: Left: a node v with its children $v_1, \dots v_k$ in the FCNS tree. Middle: annotations for left-to-right passes. v is valid if $\delta_1(\text{ann}_1(\overline{v_1}), v)$ results in an accepting state of $\mathcal{A}_1$. Right: annotations for right-to-left passes. v is valid if $\delta_2(\text{ann}_2(\overline{v_k}), v)$ is an accepting state of $\mathcal{A}_2$.

The annotation operations can be seen as streaming algorithms performing one read pass over the input and one write pass over another stream using constant space, since the annotation of a closing tag $\overline{v}$ only depends on the annotation of its right child (for left-to-right passes) or its parent (for right-to-left passes). The respective closing tag is in both cases the tag prior to v in the input stream XML(FCNS$^\perp(t)$). Remember that we consider a right-to-left pass for the annotation with ann_2.

In the following, we prove that given the annotations, Algorithm 1 and Algorithm 2 can be adapted to decide validity of the encoded form.

THEOREM 4. *There is a one-pass deterministic algorithm for* VALIDITY *with space* $O(\sqrt{N \log N})$ *and* $O(1)$ *processing time per letter when the input is given in its* FCNS$^\perp$ *encoding.*

PROOF. Append the rule $D(\perp) = \epsilon$ to the input DTD

D. Compute automaton $\mathcal{A}_1$. Compute a new XML stream with annotations ann_1 and feed this stream directly into Algorithm 1. In order to verify a node v, Algorithm 1 uses the closing tag $\overline{v_1}$ of the left child v_1 of v. Since v is valid if $\delta_1(\text{ann}_1(\overline{v_1}), v)$ is an accepting state, we only have to replace the check function used in Algorithm 1. The new check function computes $\delta_1(\text{ann}_1(\overline{v_1}), v)$ and aborts if the resulting state is not accepting.

The space requirements and the processing time per letter inherit from Algorithm 1. □

THEOREM 5. *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. Append the rule $D(\bot) = \epsilon$ to the input DTD D. Compute automata $\mathcal{A}_1$ and $\mathcal{A}_2$. Firstly, in a left-to-right pass, as in the proof of Theorem 4 we compute a new XML stream with annotations ann_1 and feed this stream directly into Algorithm 3. We adapt the check function as above.

Concerning the right-to-left pass, we compute the annotations ann_2, and feed this stream directly into Algorithm 3 interpreting opening tags as closing tags, and vice versa. Note that the annotations ann_2 are hence annotated on opening tags. Let v be a node with children $v_1, \ldots, v_k$. We have to show how Algorithm 3 can be adapted to relate the annotation of the opening tag v_k to v. When Algorithm 3 reads the closing tag $\overline{w}$ and the opening tag v_1, it pushes $(\overline{w}, v_1, \cdot, \cdot)$ on the stack, where w is the sibling of v_1 in the $\text{FCNS}^{\bot}$ encoding. Note that the subsequent tags on the stream are $v_2, v_3, \ldots v_k$. Node v_k can be identified since v_k is either a leaf or followed by a tag with label $\bot$. Hence, the stack item $(\overline{w}, v_1, \cdot, \cdot)$ can be annotated with $\text{ann}_2(v_k)$ when v_k is seen. Again by adapting the check routine, we can compute $\delta_2(\text{ann}_2(v_k), v_1)$ and abort if the result is not an accepting state.

The space requirements and the processing time per letter inherit from Algorithm 2. □

Applying the bidirectional algorithm of Theorem 5 on the encoded form $\text{XML}(\text{FCNS}^{\bot}(t))$, we obtain that validity of general trees can be decided memory efficiently in the streaming model with auxiliary streams.

COROLLARY 1. *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.*

4.4 Decoding

In the following, we present a streaming algorithm for FCNS decoding, that is, given $\text{XML}(\text{FCNS}(t))$ of some tree t, output $\text{XML}(t)$. We start with a non-streaming algorithm, Algorithm 5 performing this task.

We describe how this algorithm can be converted into a streaming algorithm. For some 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 can not be done with sublinear space in one pass (Fact 10).

Algorithm 5 offline algorithm for FCNS decoding

```
1: for i = 1 → 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
5:          write X[i]‾
6:       end if
7:    else if X[i] is a left closing tag then {See Figure 11}
8:       let p be the parent node of X[i]
9:       write p‾
10:   end if
11: end for
```

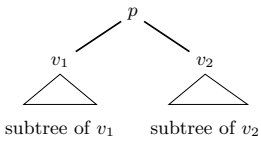

Figure 11: The main difficulty of the FCNS decoding is to write the closing tag of a node p when the closing tag of its left child is seen. This is difficult when the subtrees of v_1 and v_2 are large.

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.

4.4.1 One read-pass and two write-passes

We read blocks of size $\sqrt{N \log N}$ and execute Algorithm 5 on that block. In Lemma 6 we show 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 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 complexity derives from the following lemma demonstrating that in a block there is at most one critical left closing tag.

LEMMA 6. *Let* $X[i,j]$ *be a block. Then there is at most one left closing tag* $\overline{a}$ *with parent node* p *such that:*

$$\text{pos}(p) < i \leq \text{pos}(\overline{a}) \leq j < \text{pos}(\overline{p}). \tag{5}$$

PROOF. By 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 5 holds. Wlog we assume that $\text{pos}(p) < \text{pos}(q)$. Since $\text{pos}(p) < \text{pos}(q) < \text{pos}(\overline{a})$, q is contained in the subtree of a or $q = a$. This, however, implies that $\text{pos}(\overline{q}) \leq \text{pos}(\overline{a}) < j$ contradicting

$\mathrm{pos}(\overline{q}) > j$. □

THEOREM 6. *There is a streaming algorithm using* $O(\sqrt{N \log N})$ *space and* $O(1)$ *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* $\mathrm{XML}(\mathrm{FCNS}(t))$.

4.4.2 Logarithmic number of passes

Again, we use the offline Algorithm 5 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 write categorically *dummy symbols* on the output stream for all parent closing tags. The crux then is the following observation:

FACT 8. *Let* $\overline{c_1}_L \dots \overline{c_N}_L$ *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 2 to recover the missing tags. Instead of checking validity, once the check function is called in Algorithm 3 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 8 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 7. *There is a* $O(\log N)$*-pass streaming algorithm with space* $O(\log^2 N)$ *and* $O(\log N)$ *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.*

5. LOWER BOUNDS FOR FCNS ENCODING AND DECODING

We define a family of hard instances of length $N = \Theta(n)$ for the computation of the FCNS encoding of a tree as in Figure 12.

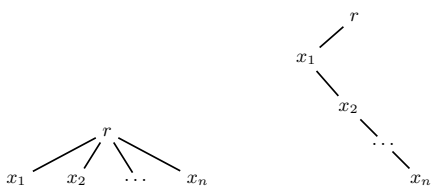

Figure 12: Left: hard instance. Right: its FCNS encoded form.

It is easy to see that computing the sequence of closing tags in the FCNS encoding requires to invert 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 $\mathrm{XML}(\mathrm{FCNS}(t)) = r_L x_{1L} x_{2R} \dots x_{nR} \overline{x_{nR}} \overline{x_{n-1R}} \dots \overline{x_{2R}} \overline{x_{1L}} \overline{r_L}$. Since the writing on the stream can only start after reading x_n, we deduce that memory space $\Omega(n)$ is required, in order to store all previous tags.

FACT 9. *Every one-pass randomized streaming algorithm for FCNS encoding with bounded error requires* $\Omega(N)$ *space.*

We conjecture that this argument can be extended as follows:

CONJECTURE 1. *Any p-passes randomized streaming algorithm for FCNS encoding with bounded error requires space* $\Omega(N/p)$.

We now define another family of hard instances of length $N = \Theta(n)$ for decoding a FCNS encoded tree as in Figure 13.

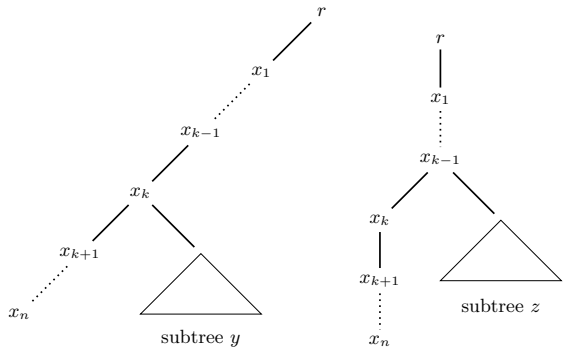

Figure 13: Left: hard instance in FCNS form, where y is any tree of size $\Theta(n)$. Right: its decoded form.

Intuitively, decoding the tree of any hard instance requires to put the full tree y into memory. Let $\mathrm{XML}(\mathrm{FCNS}(t))$ denote a hard instance which we aim to decode into $\mathrm{XML}(t)$. Then: $\mathrm{XML}(\mathrm{FCNS}(t)) = r x_{1L} \dots x_{nL} \overline{x_{nL}} \dots \overline{x_{k+1L}} Y \overline{x_{kL}} \dots \overline{x_{1L} r_L}$ and the decoded form is $\mathrm{XML}(t) = r x_1 \dots x_n \overline{x_n} \dots \overline{x_k} Z \overline{x_{k-1}} \dots \overline{x_1 r}$ where Z is the decoded form of Y. Since Z can only be written after x_k, and since x_k cannot be memorized because k was unknown until we reach Y, memory space $\Omega(n)$ is required. This argument can be easily formalized using standard information theory arguments.

FACT 10. *Every one-pass randomized streaming algorithm for FCNS decoding with bounded error requires space* $\Omega(N)$.

This argument can be extended to two-pass randomized streaming algorithms. Construct a hard instance of size $\Theta(n^2)$ by gluing n previous instances $(x^i, k^i, y^i)_{1 \le i \le n}$ as follows: instance $(x^{i+1}, k^{i+1}, y^{i+1})$ is branched to the left most leaf of instance (x^i, k^i, y^i). After the first pass, the algorithm is not able to write n closing tags of form $\overline{x^i_{k_i}}$. Therefore he needs to store them in order to write them at the second pass. This requires some formalization that we omit here.

FACT 11. *Every randomized streaming algorithm for FCNS decoding with bounded error, one pass on the input stream, and two passes on the output stream requires space* $\Omega(\sqrt{N})$.

Acknowledgements

The authors would like to thank Michel de Rougemont, who, among other things, introduced the authors to the problem of validating streaming XML documents.

6. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[2] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004.

[3] P. Beame and D.-T. Huynh-Ngoc. On the value of multiple read/write streams for approximating frequency moments. In *FOCS*, pages 499–508, 2008.

[4] P. Beame, T. Jayram, and A. Rudra. Lower bounds for randomized read/write stream algorithms. In *STOC*, pages 689–698, 2007.

[5] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. In *ACM-SIAM SODA*, pages 714–723, 2006.

[6] M. Grohe, A. Hernich, and N. Schweikardt. Randomized computations on large data sets: Tight lower bounds. In *ACM PODS*, pages 243–252, 2006.

[7] M. Grohe, A. Hernich, and N. Schweikardt. Lower bounds for processing data with few random accesses to external memory. *Journal of the ACM*, 56(3):1–16, 2009.

[8] M. Grohe, C. Koch, and N. Schweikardt. The complexity of querying external memory and streaming data. In *FCT*, pages 1–16, 2005.

[9] M. Grohe, C. Koch, and N. Schweikardt. Tight lower bounds for query processing on streaming and external memory data. *Theor. Comput. Sci.*, 380:199–217, July 2007.

[10] M. Grohe and N. Schweikardt. Lower bounds for sorting with few random accesses to external memory. In *ACM PODS*, pages 238–249, 2005.

[11] E. Kushilevitz and N. Nisan. *Communication complexity*. Cambridge University Press, 1997.

[12] F. Magniez, C. Mathieu, and A. Nayak. Recognizing well-parenthesized expressions in the streaming model. In *ACM STOC*, pages 261–270, 2010.

[13] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers Inc., 2005.

[14] F. Neven. Automata theory for xml researchers. *Sigmod Record*, 31:2002, 2002.

[15] L. Segoufin and C. Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In *ICDT*, pages 299–313, 2007.

[16] L. Segoufin and V. Vianu. Validating streaming XML documents. In *ACM PODS*, pages 53–64, 2002.

APPENDIX

A. A $\Omega(N/P)$ SPACE LOWER BOUND FOR P-PASS ALGORITHMS FOR VALIDITY

For the sake of clarity, in this section we provide 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 to problems in communication complexity [1, 2, 12]. For an introduction to communication complexity we refer the reader to [11].

Consider a player Alice holding an N bit string $x = x_1 \dots x_N$, and a player Bob holding an N bit string $y = y_1 \dots y_N$ both taken from a 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 well studied problem Set-Disjointness (DISJ).

It is well known that the randomized communication complexity with bounded two-sided error of the Set Disjointness function $R(\text{DISJ}) = \Theta(N)$. 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 $\Sigma = \{r, 0, 1\}$, the DTD D^{DISJ} 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 14.

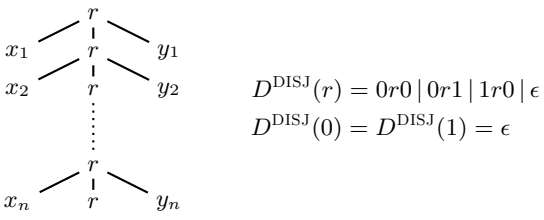

Figure 14: $t(x,y)$ is a hard instance for VALIDITY.

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

THEOREM 8. *Every p-pass randomized streaming algorithm for* VALIDITY *with bounded error 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 $rx_1\overline{x_1}rx_2\overline{x_2}\dots rx_n\overline{x_n}r$ of length $2N+2$ and executes algorithm A on this sequence using a memory of size $\text{O}(s)$. Alice send the memory of size at most s to Bob via message M_A^1 who continues algorithm A on the second half of $\text{XML}(t(x,y))$, that is $\overline{r}y_n\overline{y_n}\overline{r}\dots\overline{r}y_2\overline{y_2}\overline{r}y_1\overline{y_1}\overline{r}$ of length $2N+2$ using memory M_A^1. After execution, Bob sends the memory of size at most s back to Alice via M_B^1. This procedure is repeated at most p times.

This protocol has a total length of $\text{O}(s \cdot p)$ which we know to be $\Omega(N)$ since $R(\text{DISJ}) \in \Theta(N)$. The claim follows. □