The Streaming Complexity of Validating XML Documents

Christian Konrad
cristian.konrad@liafa.jussieu.fr
and
Frédéric Magniez
frederic.magniez@liafa.jussieu.fr

LIAFA
University Paris Diderot - Paris 7
Paris

ICDT 2012
What is XML?

- **XML document**: sequence of opening and closing tags

```
<r>
  <b>
    <a/></a>
    <a/></a>
    <c/></c>
  </b>
  <b>
    <a/></a>
    <a/></a>
    <a/></a>
  </b>
  <c/></c>
</r>
```

**Notation:** $rba\bar{a}\bar{a}\bar{a}c\bar{c}b\bar{b}\bar{b}ba\bar{a}\bar{a}b\bar{c}\bar{c}r$

- $\text{pos}(a)$, $\text{pos}(\bar{a})$: position in XML document
- $\text{depth}(a)$, $\text{depth}(\bar{a})$: depth of corresp. node

- **Depth first tree traversal**: down step gives opening tag, up step gives closing tag
Well-formedness: An XML document is well-formed iff each opening tag is closed by its corresponding closing tag

- $r\overline{a}b\overline{b}r$ is well-formed
- $\overline{r\bar{a}b\bar{b}\bar{r}}$ is not well-formed

Only well-formed documents correspond to a tree
Well-formedness: An XML document is well-formed iff each opening tag is closed by its corresponding closing tag

- $\text{raabbbar}$ is well-formed
- $\text{rabbar}$ is not well-formed

Only well-formed documents correspond to a tree

Validity: is checked wrt. a DTD (Document Type Definition)

$$
\begin{align*}
  r & \rightarrow b^* c^+ \\
  b & \rightarrow a^* c? | \epsilon \\
  a & \rightarrow \epsilon \\
  c & \rightarrow \epsilon
\end{align*}
$$

Difficulty: relate each label to labels of its children
**Well-formedness**: An XML document is well-formed iff each opening tag is closed by its corresponding closing tag

- $\overline{rabb\bar{r}}$ is well-formed
- $\overline{rab\bar{b}\bar{a}}$ is not well-formed

Only well-formed documents correspond to a tree

**Validity**: is checked wrt. a DTD (Document Type Definition)

$r \rightarrow b^* c^+$

$b \rightarrow a^* c^??|\epsilon$

$a \rightarrow \epsilon$

$c \rightarrow \epsilon$

**Difficulty**: relate each label to labels of its children
Well-formedness: An XML document is well-formed iff each opening tag is closed by its corresponding closing tag

- \( \text{raabbbar} \) is well-formed
- \( \text{rabbar} \) is not well-formed

Only well-formed documents correspond to a tree.

Validity: is checked wrt. a DTD (Document Type Definition)

\[
\begin{align*}
  r & \rightarrow b^* c^+ \\
  b & \rightarrow a^* c? | \epsilon \\
  a & \rightarrow \epsilon \\
  c & \rightarrow \epsilon
\end{align*}
\]

Difficulty: relate each label to labels of its children

not valid
Stream Computation

- **Objective**: compute some function $f(x_1, \ldots, x_n)$ given only sequential access

$x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ \ldots \ x_n$

How much RAM is required for the computation of $f$?
Stream Computation

Objective: compute some function $f(x_1, \ldots, x_n)$ given only sequential access

$x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ \ldots \ \ x_n$

How much RAM is required for the computation of $f$?
Stream Computation

- **Objective:** compute some function $f(x_1, \ldots, x_n)$ given only sequential access

  $x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ \ldots \ x_n$

  How much RAM is required for the computation of $f$?
Stream Computation

Objective: compute some function $f(x_1, \ldots, x_n)$ given only sequential access

\[ x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ \ldots \ x_n \]

How much RAM is required for the computation of $f$?

Motivation: massive data sets
- Storage on external disks, cheap sequential access
- Data streams over the internet
- XML databases can be huge
Stream Computation

**Objective:** compute some function \( f(x_1, \ldots, x_n) \) given only sequential access

\[ x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad \ldots \quad x_n \]

How much RAM is required for the computation of \( f \)?

**Motivation:** massive data sets
- Storage on external disks, cheap sequential access
- Data streams over the internet
- XML databases can be huge

**Scenarios:**
- multiple passes
- deterministic/randomized
- bidirectional
- \( \ldots \)
- auxiliary streams (external memory)
**Example**: Merge Sort with 3 streams, $O(\log N)$ passes, $O(\log N)$ space

Stream 1: $x_1 \ x_2 \ x_3 \ \ldots \ \ x_n$
Stream 2:
Stream 3:

input on stream 1
Auxiliary Streams (external memory)

**Example**: Merge Sort with 3 streams, $O(\log N)$ passes, $O(\log N)$ space

Stream 1: $x_1 \ x_2 \ x_3 \ \ldots \ \ x_n$

Stream 2: $x_1 \ x_3 \ \ldots \ \ x_{n-1}$

Stream 3: $x_2 \ x_4 \ \ldots \ \ x_n$

Copy numbers alternately onto stream 2 and stream 3
Example: Merge Sort with 3 streams, $O(\log N)$ passes, $O(\log N)$ space

Stream 1: $x_1 \ x_2 \ x_3 \ \ldots \ \ x_n$
Stream 2: $X_1 \ X_3 \ \ldots \ X_{n-1}$
Stream 3: $X_2 \ X_4 \ \ldots \ X_n$

think of numbers as sorted blocks of size 1
**Example**: Merge Sort with 3 streams, $O(\log N)$ passes, $O(\log N)$ space

Stream 1: $X_{12} \quad X_{34} \quad \ldots \quad X_{n-1,n}$
Stream 2: $X_1 \quad X_3 \quad \ldots \quad X_{n-1}$
Stream 3: $X_2 \quad X_4 \quad \ldots \quad X_n$

*merge operation*: merge blocks into blocks of size 2 onto stream 1
Example: Merge Sort with 3 streams, $O(\log N)$ passes, $O(\log N)$ space

Stream 1: $X_{12}$ $X_{34}$ ... $X_{n-1,n}$
Stream 2: $X_{12}$ $X_{56}$ ...
Stream 3: $X_{34}$ $X_{78}$ ...

copy blocks of size 2 alternately onto stream 2 and stream 3
**Example**: Merge Sort with 3 streams, $O(\log N)$ passes, $O(\log N)$ space

Stream 1: $X_{1234} \ X_{5678} \ \ldots \ X_{n-3,\ldots,n}$

Stream 2: $X_{12} \ X_{56} \ \ldots$

Stream 3: $X_{34} \ X_{78} \ \ldots$

*merge operation*: merge blocks of size 2 into blocks of size 4 onto stream 1
Auxiliary Streams (external memory)

- **Example**: Merge Sort with 3 streams, $O(\log N)$ passes, $O(\log N)$ space

Stream 1: ...  
Stream 2: ...  
Stream 3: ...

repeat this procedure until we obtain a sorted block of size $n$
**Example**: Merge Sort with 3 streams, $O(\log N)$ passes, $O(\log N)$ space

Stream 1: $X_1...n$
Stream 2: ...
Stream 3: ...

repeat this procedure until we obtain a sorted block of size $n$
Auxiliary Streams (external memory)

- **Example**: Merge Sort with 3 streams, $O(\log N)$ passes, $O(\log N)$ space

Stream 1: $X_1...n$
Stream 2: ...
Stream 3: ...

constant number of passes to double block size $\rightarrow O(\log N)$ passes
**Example:** Merge Sort with 3 streams, \( O(\log N) \) passes, \( O(\log N) \) space

Stream 1: \( X_1 \ldots n \)
Stream 2: \( \ldots \)
Stream 3: \( \ldots \)

constant number of passes to double block size \( \rightarrow O(\log N) \) passes

**Important parameters:**
- \( k(N) \) auxiliary streams
  - usually in addition to one read-only input stream
- \( p(N) \) passes
- \( s(N) \) random access space
Well-formedness: Reduction to DYCK languages

- **DYCK(k):** well-parenthesized words, $k$ types of parenthesis
  
  $$([()[]]) \in \text{DYCK}(2), ([{}]) \in \text{DYCK}(3)$$
Well-formedness: Reduction to DYCK languages

- **DYCK(k):** well-parenthesized words, \( k \) types of parenthesis
  \(([[()]]) \in \text{DYCK}(2), ([{}]) \in \text{DYCK}(3)\)

- **Well-formedness:** document well-formed if in DYCK(\( k \)):
  \[
  rba\bar{a}\bar{a}\bar{a}\bar{c}\bar{c}\bar{b}\bar{b}\bar{b}a\bar{a}a\bar{a}b\bar{c}\bar{c}r
  \]
  \[
  (r(b(a)a(a)(a)c)c)b(b)b(b(a)a(a)a)b(c)c)r
  \]
Well-formedness: Reduction to DYCK languages

- **DYCK(k):** well-parenthesized words, $k$ types of parenthesis
  
  \[
  ([()[]]) \in \text{DYCK}(2), ([{}]) \in \text{DYCK}(3)
  \]

- **Well-formedness:** document well-formed if in $\text{DYCK}(k)$:
  \[
  rba\bar{a}a\bar{a}c\bar{c}\bar{b}b\bar{b}ba\bar{a}a\bar{a}\bar{b}ccr
  
  (r(b(a)a(a)c(c)b(b)b(b(a)a(a)a)b(b)c(c))r
  
  Streaming Algorithms: Checking DYCK membership

**Theorem (F. Magniez, C. Mathieu, and A. Nayak, STOC 2010)**

- There is a **randomized** 1-pass algorithm that decides membership to $\text{DYCK}(k)$ with space $O(\sqrt{N \log k \log(N \log k)})$.

- There is a **bidirectional randomized** 2-passes algorithm that decides membership to $\text{DYCK}(k)$ with space $O((\log (N \log k))^2)$. 
Well-formedness: Reduction to $\text{DYCK}$ languages

- **$\text{DYCK} (k)$**: well-parenthesized words, $k$ types of parenthesis
  
  $([[[]]]) \in \text{DYCK}(2)$, $([{}]) \in \text{DYCK}(3)$

- **Well-formedness**: document well-formed if in $\text{DYCK} (k)$:

  $rba\bar{a}\bar{a}cc\bar{b}\bar{b}ba\bar{a}\bar{a}\bar{b}ccr$
  
  $(r(b(a)a(a)c)c)b(b)b(b(a)a(a)a)b(c)c)r$

- **Streaming Algorithms**: Checking $\text{DYCK}$ membership

---

**Theorem (F. Magniez, C. Mathieu, and A. Nayak, STOC 2010)**

- There is a randomized 1-pass algorithm that decides membership to $\text{DYCK} (k)$ with space $O(\sqrt{N \log k \log(N \log k)})$.

- There is a bidirectional randomized 2-passes algorithm that decides membership to $\text{DYCK} (k)$ with space $O((\log (N \log k))^2)$.

---

From now on: **XML documents are well-formed**
Starting Point

- **Prior works:** [Segoufin Sirangelo, 07], [Segoufin Vianu, 02] Characterization of DTDs that allow deterministic constant space validation in 1-pass
- **Upper bound:** stack based algorithm, space linear to depth of document, 1-pass deterministic
- **Lower bound:** ternary trees: any $p$ pass randomized streaming algorithm deciding validity requires $\Omega(N/p)$ space
Prior works: [Segoufin Sirangelo, 07], [Segoufin Vianu, 02] Characterization of DTDs that allow deterministic constant space validation in 1-pass

Upper bound: stack based algorithm, space linear to depth of document, 1-pass deterministic

Lower bound: ternary trees: any $p$ pass randomized streaming algorithm deciding validity requires $\Omega(N/p)$ space

DTD:

$$r \rightarrow 0r1 | 1r0 | 0r0 | \epsilon$$

$$0, 1 \rightarrow \epsilon$$

Reduction: Set-Disjointness in Communication Complexity

Christian Konrad

Streaming XML Validity
Starting Point

- **Prior works**: [Segoufin Sirangelo, 07], [Segoufin Vianu, 02]
  Characterization of DTDs that allow deterministic constant space validation in 1-pass
- **Upper bound**: stack based algorithm, space linear to depth of document, 1-pass deterministic
- **Lower bound**: ternary trees: any $p$ pass randomized streaming algorithm deciding validity requires $\Omega(N/p)$ space

\[
\begin{align*}
\text{DTD:} & \quad r \rightarrow 0r1 | 1r0 | 0r0 | \varepsilon \\
& \quad 0, 1 \rightarrow \varepsilon \\
\text{Reduction:} & \quad \text{Set-Disjointness in Communication Complexity}
\end{align*}
\]
Starting Point

- **Prior works**: [Segoufin Sirangelo, 07], [Segoufin Vianu, 02]
  Characterization of DTDs that allow deterministic constant space validation in 1-pass

- **Upper bound**: stack based algorithm, space linear to depth of document, 1-pass deterministic

- **Lower bound**: ternary trees: any p pass randomized streaming algorithm deciding validity requires $\Omega(N/p)$ space

$$\text{DTD:}$$

\[
\begin{align*}
  r & \rightarrow 0r1 | 1r0 | 0r0 | \epsilon \\
  0, 1 & \rightarrow \epsilon
\end{align*}
\]

- **Reduction**: Set-Disjointness in Communication Complexity
Main Result

Theorem

There is a bidirectional $O(\log N)$-pass deterministic streaming algorithm for validity of arbitrary XML files and arbitrary DTDs with space $O(\log^2 N)$ and 3 auxiliary streams.
Main Result

**Theorem**

There is a bidirectional $O(\log N)$-pass deterministic streaming algorithm for validity of arbitrary XML files and arbitrary DTDs with space $O(\log^2 N)$ and 3 auxiliary streams.

**Steps:**

1. **Using 3 aux. streams, $O(\log N)$ space, $O(\log N)$ passes:**
   Compute the FCNS (First-Child-Next-Sibling) encoding of the original document (encoding as a binary tree)

2. **Using 2 bidirectional passes, $O(\log^2 N)$ space:**
   Check validity based on this binary encoding
   Algorithm inspired by algorithm for checking validity of binary trees
Theorem

There is a one-pass deterministic algorithm using $O(\sqrt{N \log N})$ space for checking validity of binary trees.

Conjecture: there is no one-pass algorithm using $o(\sqrt{N \log N})$ space even when randomization is allowed.

Theorem

There is a bidirectional two-passes deterministic algorithm using $O(\log^2 N)$ space for checking validity of binary trees.
Two-passes Algorithm for Validity of binary Trees

Lemma (1)

There is a one-pass deterministic algorithm using $O(\log^2 N)$ space that verifies validity of all nodes which have a left subtree that is at least as large as its right subtree.
Lemma (1)

There is a one-pass deterministic algorithm using \(O(\log^2 N)\) space that verifies validity of all nodes which have a left subtree that is at least as large as its right subtree.

Theorem

There is a bidirectional two-passes deterministic algorithm using \(O(\log^2 N)\) space for validity.
**Two-passes Algorithm for Validity of binary Trees**

**Lemma (1)**

There is a one-pass deterministic algorithm using \( O(\log^2 N) \) space that verifies validity of all nodes which have a left subtree that is at least as large as its right subtree.

**Theorem**

There is a bidirectional two-passes deterministic algorithm using \( O(\log^2 N) \) space for validity.

**Proof:**

1. Run algorithm of Lemma 1
Lemma (1)

There is a one-pass deterministic algorithm using $O(\log^2 N)$ space that verifies validity of all nodes which have a left subtree that is at least as large as its right subtree.

Theorem

There is a bidirectional two-passes deterministic algorithm using $O(\log^2 N)$ space for validity.

Proof:

1. Run algorithm of Lemma 1
2. Run algorithm of Lemma 1 on the input stream read from right to left interpreting opening tags as closing tags and vice versa.
**Goal:** for all internal nodes $p$: relate $p$ to its children $u$, $v$ via check($p$, $u$, $v$)

![Binary Tree Diagram]

...pu ....... uv ........ vp ...

- *Strategies:* store either $p$ until $uv$ arrives, or throw $p$ away and store $uv$ until $p$ arrives
Binary Trees - general ideas

- **Goal:** for all internal nodes $p$: relate $p$ to its children $u, v$ via
  \[
  \text{check}(p, u, v)
  \]

- Two chances for verification:
  - Top down using $p, \bar{u}, \bar{v}$

\[
\begin{array}{c}
p \\
\uparrow & \uparrow \\
u & v \\
\text{subtree of } u & \text{subtree of } v \\
\vdots pu \ldots \bar{u}v \ldots \bar{v}p \ldots
\end{array}
\]
**Binary Trees - general ideas**

**Goal:** for all internal nodes $p$: relate $p$ to its children $u, v$ via \( \text{check}(p, u, v) \)

\[ p \quad u \quad v \]

subtree of $u$   subtree of $v$

\[
\ldots pu \quad \ldots \bar{u}v \quad \ldots \bar{v}p \ldots
\]

**Two chances for verification:**
- Top down using $p, \bar{u}, v$
- Bottom up using $\bar{u}, v, \bar{p}$
**Binary Trees - general ideas**

- **Goal:** for all internal nodes $p$: relate $p$ to its children $u, v$ via check($p, u, v$)

```
  p
 / \
 u   v
    / \    / \
   /   / \  /   / \  
  /   /   / \  /   /   \  
 /   /   /   \ /   /   \   
```

subtree of $u$  subtree of $v$

$\ldots pu \ldots \bar{uv} \ldots \bar{V} \bar{p} \ldots$

- Two chances for verification:
  - Top down using $p, \bar{u}, v$
  - Bottom up using $\bar{u}, v, \bar{p}$

- $\bar{u}, v$ is used for verification in any case

**Strategies:** store either $p$ until $\bar{uv}$ arrives, or throw $p$ away and store $\bar{uv}$ until $\bar{p}$ arrives
1st idea: Start with stack algorithm doing bottom-up verifications

\[ u \quad v \]

<table>
<thead>
<tr>
<th>subtree of ( u )</th>
<th>subtree of ( v )</th>
</tr>
</thead>
</table>

\[ \ldots pu \ldots \overline{uv} \ldots \overline{vp} \ldots \]

- Ignore opening tags of parent nodes
1st idea: Start with stack algorithm doing bottom-up verifications

Ignore opening tags of parent nodes
Push children information on a stack:
**1st idea:** Start with stack algorithm doing bottom-up verifications

![Diagram of a tree structure]

- Ignore opening tags of parent nodes
- Push children information on a stack:

  - \(u_1, v_1\)
  - \(u_2, v_2\)
  - \(u_3, v_3\)
  - \(u_4, v_4\)

- Verify when going up
1st idea: Start with stack algorithm doing bottom-up verifications

- Stack of linear size
- Verification of all nodes

Ignore opening tags of parent nodes
Push children information on a stack:
Verifying nodes with larger left subtree

2nd idea: Reduce stack to \( \log(n) \) elements: remove children \( \overline{uv} \) from stack whose parents’ node has a smaller left subtree than its right subtree.

\[
\begin{array}{c}
\vdots \\
\overline{u}_2, \overline{v}_2 \\
\overline{u}_1, \overline{v}_1 \\
\vdots \\

\end{array}
\]

\[
\begin{array}{c}
p \\
\overline{u}_1 \\
\overline{v}_1 \\
\overline{u}_2 \\
\overline{v}_2 \\
\vdots \\
\end{array}
\]

c: current item in stream

XML stream:

\[
pos(\overline{u}_1) \quad pos(\overline{u}_2) \quad pos(c)
\]
Verifying nodes with larger left subtree

**2nd idea:** Reduce stack to \( \log(n) \) elements: remove children \( \overline{uv} \) from stack whose parents’ node has a smaller left subtree than its right subtree

- XML stream:
  - \( \text{pos} \left( \overline{u_1} \right) \)  \( \text{pos}(u_2) \)  \( \text{pos}(c) \)

  \( \bullet \) \( \text{pos}(c) - \text{pos}(u_2) \leq \) size of right subtree of \( q \)
2nd idea: Reduce stack to $\log(n)$ elements: remove children $\overline{uv}$ from stack whose parents’ nodes have a smaller left subtree than its right subtree.

XML stream:

- $\text{pos}(\overline{u_1})$ to $\text{pos}(\overline{u_2})$ to $\text{pos}(c)$

- $\text{pos}(c) - \text{pos}(\overline{u_2}) \leq \text{size of right subtree of } q$
- $\text{pos}(\overline{u_2}) - \text{pos}(\overline{u_1}) \geq \text{size of left subtree of } q$
Verifying nodes with larger left subtree

2nd idea: Reduce stack to $\log(n)$ elements: remove children $\bar{uv}$ from stack whose parents’ node has a smaller left subtree than its right subtree

![Diagram of tree structure]

- $p$
- $u_1$
- $v_1$
- $u_2$
- $v_2$
- $c$

XML stream:

- pos($u_1$)
- pos($u_2$)
- pos($c$)

- $\text{pos}(c) - \text{pos}(u_2) \leq \text{size of right subtree of } q$
- $\text{pos}(u_2) - \text{pos}(u_1) \geq \text{size of left subtree of } q$

Deletion rule: delete if $\text{pos}(c) - \text{pos}(u_2) > \text{pos}(u_2) - \text{pos}(u_1)$
Verifying nodes with larger left subtree

2nd idea: Reduce stack to $\log(n)$ elements: remove children $uv$ from stack whose parents’ node has a smaller left subtree than its right subtree

\[ \begin{array}{c}
\vdots \\
u_2, v_2 \\
u_1, v_1 \\
\vdots \\
\end{array} \]

\[ p \]

\[ u_1 \quad v_1 \]

\[ u_2 \quad v_2 \]

\[ q \]

\[ c: \text{current item in stream} \]

XML stream:

\[ \text{pos}(u_1) \quad \text{pos}(u_2) \quad \text{pos}(c) \]

- $\text{pos}(c) - \text{pos}(u_2) \leq \text{size of right subtree of } q$
- $\text{pos}(u_2) - \text{pos}(u_1) \geq \text{size of left subtree of } q$

Deletion rule: delete if $\text{pos}(c) - \text{pos}(u_2) > \text{pos}(u_2) - \text{pos}(u_1)$

$\rightarrow$ In doing so stack is of size at most $\log N$. 
Lemma: Stack is of size at most $\log(N)$.

Proof:

- **Deletion rule:** $\text{pos}(c) - \text{pos}(u_i) > \text{pos}(u_i) - \text{pos}(u_{i-1})$
- $u_i, v_i$ remains on stack: $\Rightarrow$ deletion rule does not apply

<table>
<thead>
<tr>
<th>$u_{max}, v_{max}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdots$</td>
</tr>
<tr>
<td>$u_2, v_2$</td>
</tr>
<tr>
<td>$u_1, v_1$</td>
</tr>
</tbody>
</table>

\[
\text{pos}(u_i) > \frac{\text{pos}(c) + \text{pos}(u_{i-1})}{2}. \\
\text{pos}(u_2) \geq \left\lfloor \frac{\text{pos}(c)}{2} \right\rfloor, \\
\text{pos}(u_3) \geq \left\lfloor \frac{3\text{pos}(c)}{4} \right\rfloor, \\
\ldots \]

- This leaves only space for $\log(\text{pos}(c))$ elements
First-Child-Next-Sibling encoding

1. Original document

2. Keep edges to first children

3. Insert edges connecting children

4. FCNS encoding
First-Child-Next-Sibling encoding

**Transformation:** For each node in original document:
- **First child:** becomes *left child* of that node
- **Next Sibling:** becomes *right child* of that node

**Annotation:** tags in FCNS encoding are annotated *left/right*
Validation is easier given the FCNS encoding

**Original document:** tags of children of \( v \) scattered

\[
\begin{align*}
v & \quad v_1 \quad v_2 \quad \cdots \quad v_k \\
t_1 & \quad t_2 \quad t_k
\end{align*}
\]

\[
\begin{align*}
\tilde{v} & \quad \tilde{v}_1 \quad \tilde{v}_2 \quad \cdots \quad \tilde{v}_k \\
\tilde{t}_1' & \quad \tilde{t}_2' \quad \tilde{t}_k'
\end{align*}
\]
Validation is easier given the FCNS encoding

- **Original document**: tags of children of $\nu$ scattered
  $$\ldots \nu \nu_1 \nu_2 \nu_3 \ldots \nu_{k-1} \nu_k \nu \ldots$$

- **FCNS encoding**: $\overline{\nu_k \nu_{k-1} \ldots \nu_2 \nu_1}$ appears as substring
  $$\ldots \nu \nu_1 \nu_2 \nu_k \ldots \overline{\nu_k \nu_{k-1} \ldots \nu_2 \nu_1} \nu \ldots$$
Reusing binary tree validation algorithm

**FCNS encoding:**

Left-to-Right pass:
- compress subsequence \( \overline{v_k} \ldots \overline{v_1} \) via an automaton \( A_L \) constructed from initial DTD into a state
- annotate \( \overline{v_1} \) with that state
- binary tree validation algorithm relates state to label of parent

Right-to-Left pass:
- compress subsequence \( v_1 \ldots v_k \) via an automaton \( A_R \) constructed from initial DTD into a state
- if binary tree algorithm pushed \( v_1 \) onto stack, annotate stack element by this state
- binary tree validation algorithm relates state to label of parent
Computing the FCNS encoding: reordering of XML tags and annotation

Algorithm:

1. compute sequence of opening tags with annotations
   sequences of opening tags coincide
2. compute sequence of closing tags with annotations
   start with sequence of opening tags, interpret them as closing tags, and reorder them via a modified merge sort
3. merge these sequences
Computing the FCNS encoding:

Algorithm:
1. compute sequence of opening tags with annotations
   sequences of opening tags coincide
2. compute sequence of closing tags with annotations
   start with sequence of opening tags, interpret them as closing tags, and reorder them via a modified merge sort
3. merge these sequences
Computing the FCNS encoding

Algorithm:
1. Compute sequence of opening tags with annotations
   sequences of opening tags coincide
2. Compute sequence of closing tags with annotations
   start with sequence of opening tags, interpret them as closing tags, and reorder them via a modified merge sort
3. Merge these sequences

Computing the FCNS encoding: reordering of XML tags and annotation
We have:

- One pass, $O(\sqrt{N \log N})$ space for two-ranked trees
- Two bidirectional passes, $O(\log^2 N)$ space for two-ranked trees
- $O(\log N)$ passes, 3 aux. streams, $O(\log^2 N)$ space for arbitrary trees

Open Problems:

- Lower bound: optimality of one pass algorithm for binary trees
- Lower bound: $\Omega(\log(N))$ passes are required for unranked trees when using sublinear space and a constant number of auxiliary streams
- Other membership problems: $\text{DYCK}(k) \subset \text{Visibly Pushdown languages} \subset \text{deterministic context free languages} \subset \text{context free languages}$