The Streaming Complexity of Validating XML Documents

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

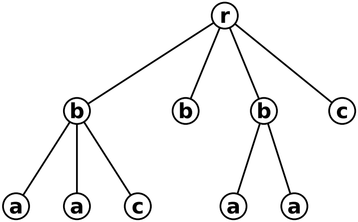
LIAFA
University Paris Diderot - Paris 7
Paris

ICDT 2012

What is XML?

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

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



Notation: $rba\bar{a}a\bar{a}c\bar{c}\bar{b}b\bar{b}ba\bar{a}a\bar{a}\bar{b}c\bar{c}\bar{r}$
$\mathrm{pos}(a), \mathrm{pos}(\bar{a})$: position in XML document
$\mathrm{depth}(a), \mathrm{depth}(\bar{a})$: depth of corresp. node

- **Depth first tree traversal**: down step gives opening tag, up step gives closing tag

Well-formedness and Validity

Well-formedness: An XML document is well-formed iff each opening tag is closed by its corresponding closing tag

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

Only well-formed documents correspond to a tree

Well-formedness and Validity

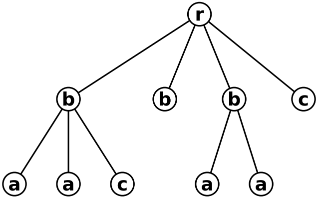
Well-formedness: An XML document is well-formed iff each opening tag is closed by its corresponding closing tag

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

Only well-formed documents correspond to a tree

Validity: is checked wrt. a DTD (**D**ocument **T**ype **D**efinition)

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



Difficulty: relate each label to labels of its children

Well-formedness and Validity

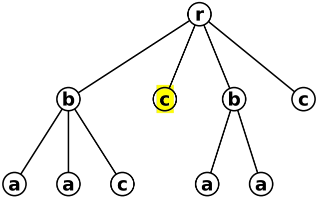
Well-formedness: An XML document is well-formed iff each opening tag is closed by its corresponding closing tag

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

Only well-formed documents correspond to a tree

Validity: is checked wrt. a DTD (**D**ocument **T**ype **D**efinition)

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



Difficulty: relate each label to labels of its children

Well-formedness and Validity

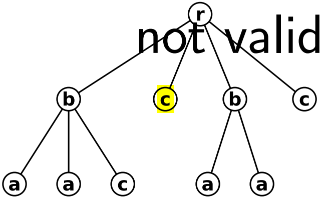
Well-formedness: An XML document is well-formed iff each opening tag is closed by its corresponding closing tag

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

Only well-formed documents correspond to a tree

Validity: is checked wrt. a DTD (**D**ocument **T**ype **D**efinition)

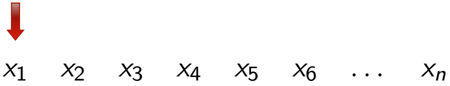
$$\begin{aligned} r &\rightarrow b^*c^+ \\ b &\rightarrow a^*c?|\epsilon \\ a &\rightarrow \epsilon \\ c &\rightarrow \epsilon \end{aligned}$$



Difficulty: relate each label to labels of its children

Stream Computation

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

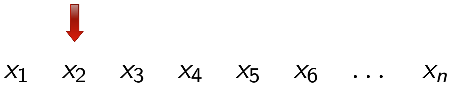


$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad \ldots \quad x_n$

How much RAM is required for the computation of f?

Stream Computation

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

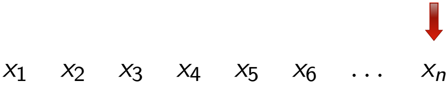


$$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad \dots \quad x_n$$

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

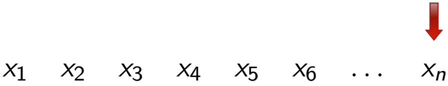


$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad \ldots \quad x_n$

How much RAM is required for the computation of f?

Stream Computation

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



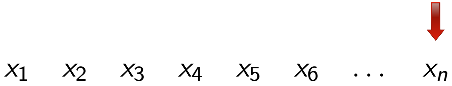
$$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad \dots \quad x_n$$

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

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
  - ...
  - **auxiliary streams** (external memory)

Auxiliary Streams (external memory)

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

Stream 1: $x_1 \quad x_2 \quad x_3 \quad \dots \qquad\qquad x_n$
Stream 2:
Stream 3:

input on stream 1

Auxiliary Streams (external memory)

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

Stream 1: $x_1 \quad x_2 \quad x_3 \quad \ldots \quad x_n$
Stream 2: $x_1 \quad x_3 \quad \ldots \quad x_{n-1}$
Stream 3: $x_2 \quad x_4 \quad \ldots \quad x_n$

copy numbers aternately onto stream 2 and stream 3

Auxiliary Streams (external memory)

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

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

think of numbers as sorted blocks of size 1

Auxiliary Streams (external memory)

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

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

merge operation: merge blocks into blocks of size 2 onto stream 1

Auxiliary Streams (external memory)

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

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

copy blocks of size 2 alternately onto stream 2 and stream 3

Auxiliary Streams (external memory)

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

Stream 1: X_{1234} X_{5678} $\dots$ $X_{n-3,\dots,n}$
Stream 2: X_{12} X_{56} $\dots$
Stream 3: X_{34} X_{78} $\dots$

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, $\mathrm{O}(\log N)$ passes, $\mathrm{O}(\log N)$ space

Stream 1: $\ldots$
Stream 2: $\ldots$
Stream 3: $\ldots$

repeat this procedure until we obtain a sorted block of size n

Auxiliary Streams (external memory)

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

Stream 1: $X_{1\ldots n}$
Stream 2: $\ldots$
Stream 3: $\ldots$

repeat this procedure until we obtain a sorted block of size n

Auxiliary Streams (external memory)

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

Stream 1: $X_{1\dots n}$
Stream 2: $\dots$
Stream 3: $\dots$

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

Auxiliary Streams (external memory)

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

Stream 1: $X_{1\dots n}$
Stream 2: $\dots$
Stream 3: $\dots$

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

  $([()[]]) \in \mathrm{DYCK}(2)$, $([\{\}]) \in \mathrm{DYCK}(3)$

Well-formedness: Reduction to DYCK languages

- **DYCK(k):** well-parenthesized words, *k* types of parenthesis

$$([()[]]) \in \mathrm{DYCK}(2),\ ([\{\}]) \in \mathrm{DYCK}(3)$$

- **Well-formedness:** document well-formed if in DYCK(k):

$$r b a \overline{a} a \overline{a} c \overline{c} \overline{b} b \overline{b} b a \overline{a} a \overline{a} \overline{b} c \overline{c} \overline{r}$$

$$(_r (_b (_a)_a (_a)_a (_c)_c)_b (_b)_b (_b (_a)_a (_a)_a)_b (_c)_c)_r$$

Well-formedness: Reduction to DYCK languages

- **DYCK(k):** well-parenthesized words, k types of parenthesis

$$([()[]]) \in \mathrm{DYCK}(2),\ (\{\}]) \in \mathrm{DYCK}(3)$$

- **Well-formedness:** document well-formed if in DYCK(k):

$$rba\bar{a}a\bar{a}c\bar{c}\bar{b}b\bar{b}ba\bar{a}a\bar{a}\bar{b}c\bar{c}\bar{r}$$

$$(_r(_b(_a)_a(_a)_a(_c)_c)_b(_b)_b(_b(_a)_a(_a)_a)_b(_c)_c)_r$$

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

Well-formedness: Reduction to DYCK languages

- **DYCK(k):** well-parenthesized words, k types of parenthesis

$$([()[]]) \in \mathrm{DYCK}(2),\ ([\{\}]) \in \mathrm{DYCK}(3)$$

- **Well-formedness:** document well-formed if in DYCK(k):

$$rba\overline{a}a\overline{a}c\overline{c}\overline{b}b\overline{b}ba\overline{a}a\overline{a}\overline{b}c\overline{c}\overline{r}$$
$$(_r(_b(_a)_a(_a)_a(_c)_c)_b(_b)_b(_b(_a)_a(_a)_a)_b(_c)_c)_r$$

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

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

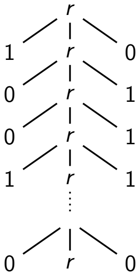
Starting Point

- **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

DTD:

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

$0, 1 \rightarrow \epsilon$



$$r1\overline{1}r0\overline{0}r0\overline{0}r1\overline{1}r\ldots0\overline{0}r\overline{r}0\overline{0}\ldots\overline{r}1\overline{1}\overline{r}1\overline{1}\overline{r}1\overline{1}\overline{r}0\overline{0}\overline{r}$$

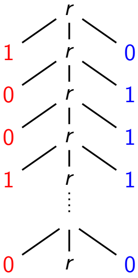
Starting Point

- **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

DTD:

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

$0, 1 \rightarrow \epsilon$



$r1\overline{1}r0\overline{0}r0\overline{0}r1\overline{1}r\dots0\overline{0}r\overline{r}0\overline{0}\dots\overline{r}1\overline{1}\overline{r}1\overline{1}\overline{r}1\overline{1}\overline{r}0\overline{0}\overline{r}$

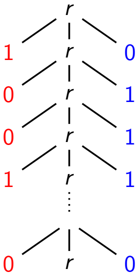
Starting Point

- **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

DTD:

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

$0, 1 \rightarrow \epsilon$



$$r1\bar{1}r0\bar{0}r0\bar{0}r1\bar{1}r\dots0\bar{0}r\bar{r}0\bar{0}\dots\bar{r}1\bar{1}\bar{r}1\bar{1}\bar{r}1\bar{1}\bar{r}0\bar{0}\bar{r}$$

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

Main Result

Theorem

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

Main Result

Theorem

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

Steps:

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

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

Binary Trees - Results

Theorem

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

Two-passes Algorithm for Validity of binary Trees

Lemma (1)

There is a one-pass **deterministic** *algorithm using* $\mathrm{O}(\log^2 N)$ *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* $\mathrm{O}(\log^2 N)$ *space for validity.*

Two-passes Algorithm for Validity of binary Trees

Lemma (1)

There is a one-pass **deterministic** *algorithm using* $\mathrm{O}(\log^2 N)$ *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* $\mathrm{O}(\log^2 N)$ *space for validity.*

Proof:

1. Run algorithm of Lemma 1

Two-passes Algorithm for Validity of binary Trees

Lemma (1)

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

Theorem

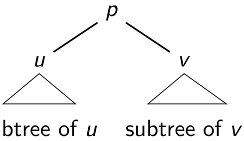
There is a bidirectional two-passes **deterministic** *algorithm using* $\mathrm{O}(\log^2 N)$ *space for validity.*

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.

Binary Trees - general ideas

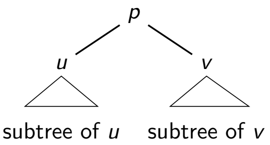
- **Goal:** for all internal nodes p: relate p to its children u, v via check(p, u, v)



$$\dots pu \dots\dots \overline{u}v \dots\dots \overline{vp} \dots$$

Binary Trees - general ideas

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

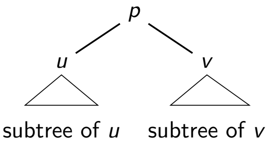


$$\ldots pu \ldots\ldots \overline{u}v \ldots\ldots \overline{v}\overline{p} \ldots$$

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

Binary Trees - general ideas

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

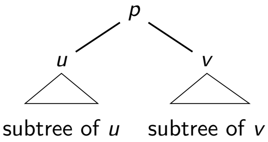


$\ldots pu \ldots\ldots \overline{u}v \ldots\ldots \overline{v}\,\overline{p} \ldots$

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

Binary Trees - general ideas

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



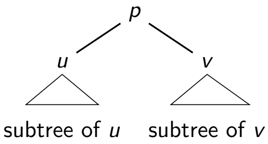
$$\dots pu \dots\dots \overline{u}v \dots\dots \overline{v}\ \overline{p}\ \dots$$

- Two chances for verification:
  - Top down using $p, \overline{u}, v$
  - Bottom up using $\overline{u}, v, \overline{p}$
- $\overline{u}, v$ is used for verification in any case

Strategies: store either p until $\overline{u}v$ arrives, or throw p away and store $\overline{u}v$ until $\overline{p}$ arrives

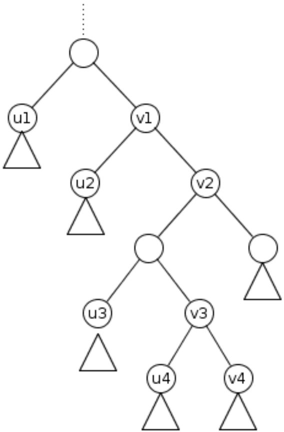
Stack Algorithm

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



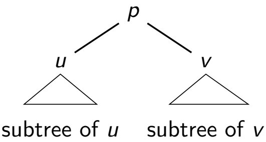
$\ldots pu \ldots\ldots \overline{u}v \ldots\ldots \overline{v}p \ldots$

- Ignore opening tags of parent nodes



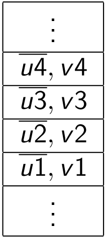
Stack Algorithm

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



$\ldots pu \ldots\ldots \overline{u}v \ldots\ldots \overline{v}\overline{p} \ldots$

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



$\vdots$

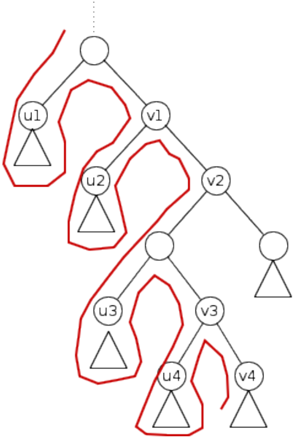
$\overline{u4}, v4$

$\overline{u3}, v3$

$\overline{u2}, v2$

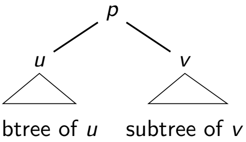
$\overline{u1}, v1$

$\vdots$



Stack Algorithm

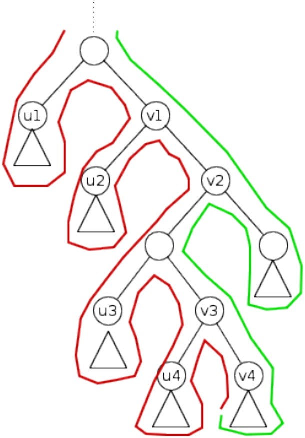
1st idea: Start with stack algorithm doing bottom-up verifications



$\dots pu \dots\dots \overline{u}v \dots\dots \overline{vp}\dots$

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

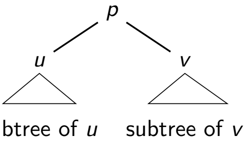
$\vdots$
$\overline{u4}, v4$
$\overline{u3}, v3$
$\overline{u2}, v2$
$\overline{u1}, v1$
$\vdots$



- Verify when going up

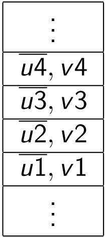
Stack Algorithm

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



$\ldots pu \ldots\ldots \overline{u}v \ldots\ldots \overline{v}\overline{p}\ldots$

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



$\vdots$

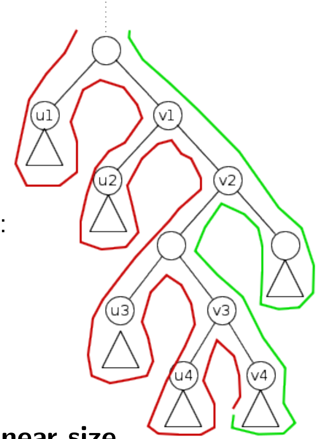
$\overline{u4}, v4$

$\overline{u3}, v3$

$\overline{u2}, v2$

$\overline{u1}, v1$

$\vdots$

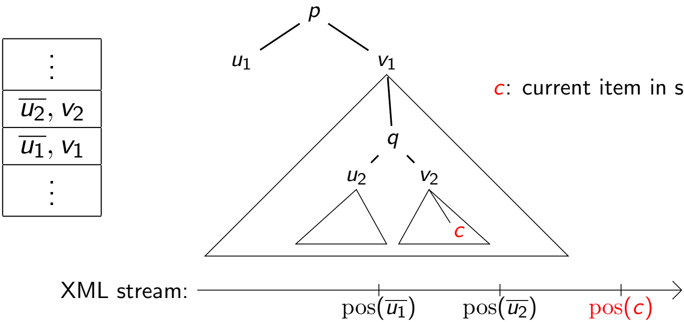


- **Stack of linear size**
- **Verification of all nodes**

- Verify when going up

Verifying nodes with larger left subtree

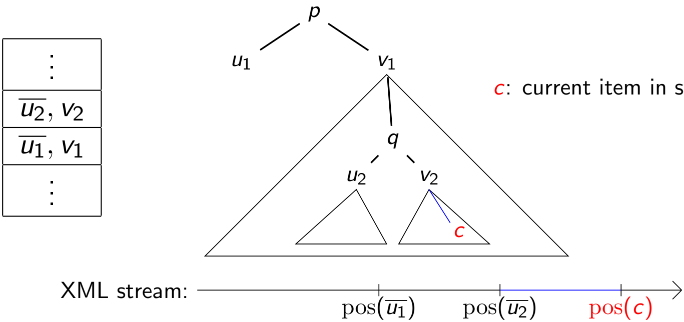
2nd idea: Reduce stack to $\log(n)$ elements: remove children $\overline{u}v$ from stack whose parents' node has a smaller left subtree than its right subtree



c: current item in stream

Verifying nodes with larger left subtree

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

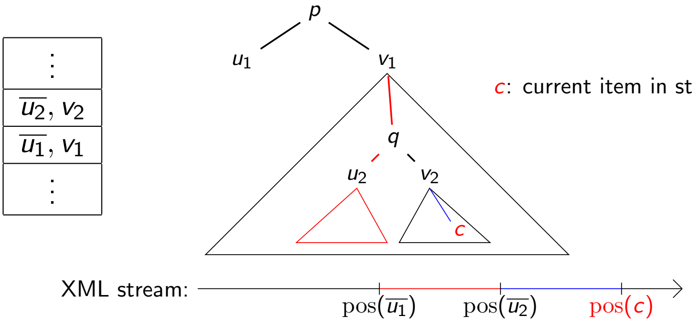


c: current item in stream

- $\mathrm{pos}(c)$ - $\mathrm{pos}(\overline{u_2}) \leq$ size of right subtree of q

Verifying nodes with larger left subtree

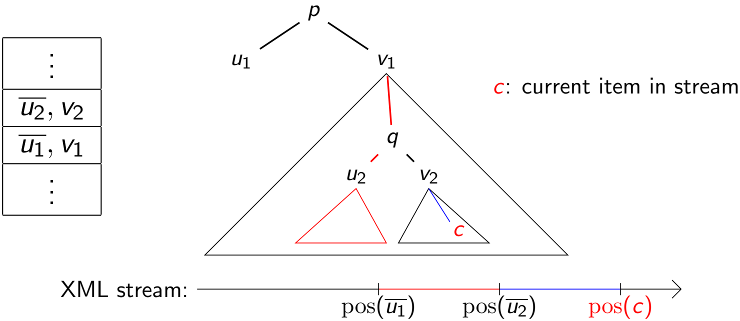
2nd idea: Reduce stack to $\log(n)$ elements: remove children $\overline{u}v$ from stack whose parents' node has a smaller left subtree than its right subtree



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

Verifying nodes with larger left subtree

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

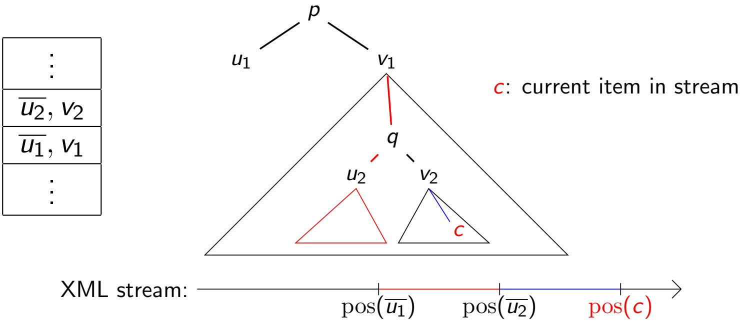


- $\mathrm{pos}(c)$ - $\mathrm{pos}(\overline{u_2}) \leq$ size of right subtree of q
- $\mathrm{pos}(\overline{u_2}) - \mathrm{pos}(\overline{u_1}) \geq$ size of left subtree of q

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

Verifying nodes with larger left subtree

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



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

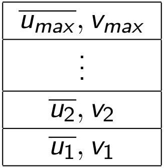
Deletion rule: delete if $\mathrm{pos}(c) - \mathrm{pos}(\overline{u_2}) > \mathrm{pos}(\overline{u_2}) - \mathrm{pos}(\overline{u_1})$
→ In doing so stack is of size at most $\log N$.

Verifying nodes with larger left subtree (3)

Lemma: Stack is of size at most $\log(N)$.

Proof:

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



$\overline{u_{max}}, v_{max}$

$\vdots$

$\overline{u_2}, v_2$

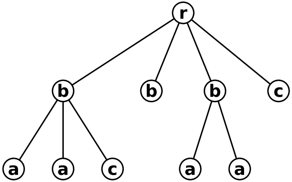
$\overline{u_1}, v_1$

$$\mathrm{pos}(\overline{u_i}) > \frac{\mathrm{pos}(c) + \mathrm{pos}(\overline{u_{i-1}})}{2}.$$

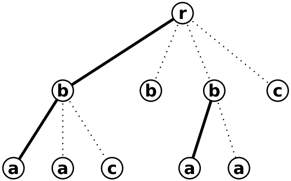
$$\begin{aligned} \mathrm{pos}(\overline{u_2}) &\geq \left\lceil \frac{\mathrm{pos}(c)}{2} \right\rceil, \\ \mathrm{pos}(\overline{u_3}) &\geq \left\lceil \frac{3\mathrm{pos}(c)}{4} \right\rceil, \\ &\dots \end{aligned}$$

- This leaves only space for $\log(\mathrm{pos}(c))$ elements $\square$

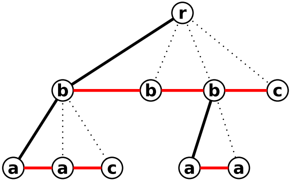
First-Child-Next-Sibling encoding



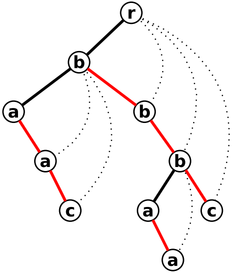
1. Original document



2. Keep edges to first children

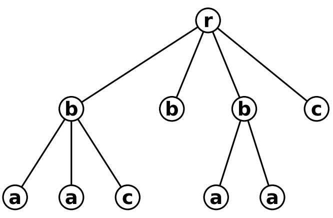


3. Insert edges connecting children

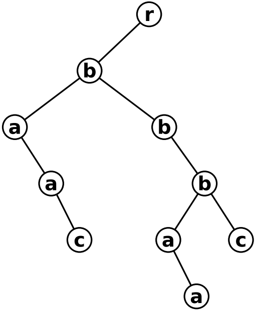


4. FCNS encoding

First-Child-Next-Sibling encoding



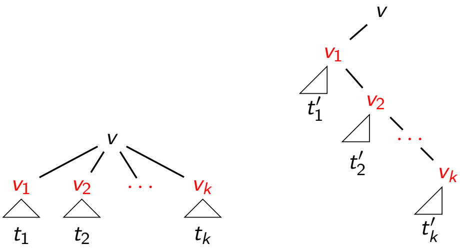
original tree



FCNS encoding

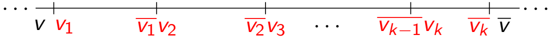
- **Transformation:** For each node in original document:
  - **First child:** becomes *left child* of that node
  - **Next Sibling:** becomes *right child* of that node
- **Annotation:** tags in FCNS encoding are annotated *left*/*right*

Validation is easier given the FCNS encoding

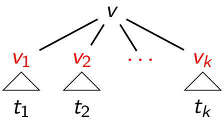


original document FCNS encoding

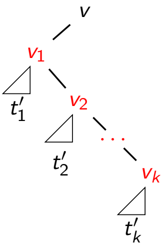
- **Original document:** tags of children of v scattered



Validation is easier given the FCNS encoding

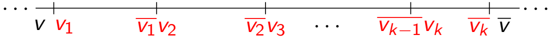


original document

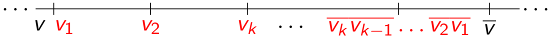


FCNS encoding

- **Original document:** tags of children of v scattered

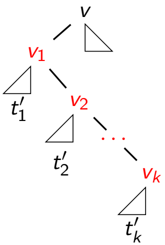


- **FCNS encoding:** $\overline{v_k v_{k-1}} \dots \overline{v_2 v_1}$ appears as substring



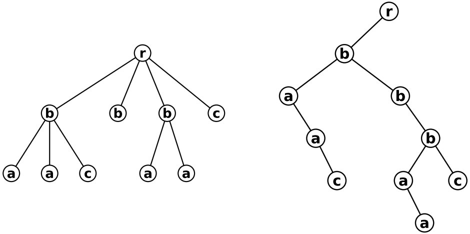
Reusing binary tree validation algorithm

FCNS encoding:



- **Left-to-Right pass:**
  - compress subsequence $\overline{v_k} \dots \overline{v_1}$ 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 $v_1 \dots v_k$ 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

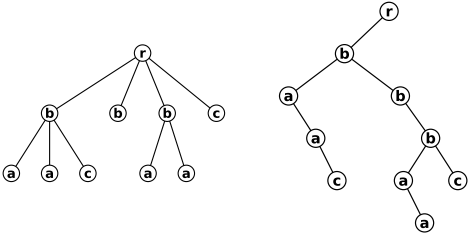
Computing the FCNS encoding



$rba\overline{a}a\overline{a}c\overline{c}\overline{b}b\overline{b}ba\overline{a}a\overline{a}\overline{b}c\overline{c}\overline{r}$ $rbaac\overline{caa}bbaa\overline{aa}c\overline{c}\overline{bbb}\overline{r}$

- **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**

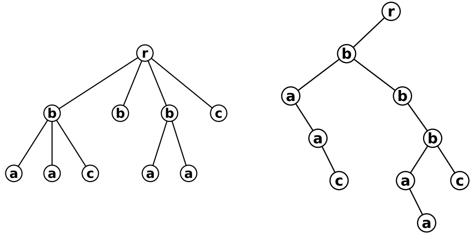
Computing the FCNS encoding



$rba\bar{a}a\bar{a}c\bar{c}\bar{b}b\bar{b}ba\bar{a}a\bar{a}\bar{b}c\bar{c}\bar{r}$ $rbaac\bar{c}\bar{a}\bar{a}bbaa\bar{a}\bar{a}c\bar{c}\bar{b}\bar{b}\bar{b}\bar{r}$

- **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



$rba{\color{red}\overline{a}}a{\color{red}\overline{a}}c{\color{red}\overline{c}}\overline{b}b\overline{b}ba{\color{red}\overline{a}}a{\color{red}\overline{a}}\overline{b}c\overline{c}\overline{r}$ $rbaac{\color{red}\overline{caa}}bbaa{\color{red}\overline{aa}}c\overline{c}\overline{bbb}\overline{r}$

- **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**

Conclusion

We have:

- One pass, $\mathrm{O}(\sqrt{N \log N})$ space for two-ranked trees
- Two bidirectional passes, $\mathrm{O}(\log^2 N)$ space for two-ranked trees
- $\mathrm{O}(\log N)$ passes, 3 aux. streams, $\mathrm{O}(\log^2 N)$ 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: DYCK(k) $\subset$ Visibly Pushdown languages $\subset$ deterministic context free languages $\subset$ context free languages