XML and Databases
XPath evaluation (3)

Kim.Nguyen@nicta.com.au

Week 10
Recap from last week

1 Using automata to run queries in streaming
Recap from last week

1. Using automata to run queries in streaming
2. Handling filters with upward axes
Recap from last week

1. Using automata to run queries in streaming
2. Handling filters with upward axes
Recap from last week

1. Using automata to run queries in streaming
2. Handling filters with upward axes

Today

- How to add preceding-sibling/following-sibling?
- What data structures to use for automata?
Following-sibling

Fundamentally, not very different from child!
In a pre-order traversal:

child : From a node, go firstChild then nextSibling, ..., nextSibling until NULL is found

following-sibling : From a node, go nextSibling, ..., nextSibling until NULL is found
Following-sibling

Fundamentally, not very different from child!

In a pre-order traversal:

```plaintext
child : From a node, go firstChild then nextSibling, ..., nextSibling until NULL is found
following-sibling : From a node, go nextSibling, ..., nextSibling until NULL is found
```

What does it mean in terms of automata?
Following-sibling

Fundamentally, not very different from child!

In a pre-order traversal:

child: From a node, go `firstChild` then `nextSibling`, ...,
nextSibling until `NULL` is found

following-sibling: From a node, go `nextSibling`, ...,
nextSibling until `NULL` is found

What does it mean in terms of automata?

Add a new \textit{kind} of transition
//a/b//d

When we evaluate an automaton we can perform two kinds of transitions:

• When doing a first child move, we take black transitions (Down)
• When doing a next sibling move, we take red transitions (Right)
When we evaluate an automaton we can perform two kinds of transitions:

- When doing a first child move, we take black transitions (Down)
- When doing a next sibling move, we take red transitions (Right)

Example

//a/b//d

1 \rightarrow a \rightarrow b \rightarrow d \rightarrow 4
When we evaluate an automaton we can perform two kinds of transitions:

- When doing a first child move, we take black transitions (Down)
- When doing a next sibling move, we take red transitions (Right)

//a//b//d

//a/following-sibling::b//d?
When we evaluate an automaton we can perform two kinds of transitions:

- When doing a first child move, we take black transitions (Down).
- When doing a next sibling move, we take red transitions (Right).

Example:

```
//a/b//d
```

```
//a/following-sibling::b//d?
```

Diagram:

```
1 2 3 4
```

```
1 2 3 4
```

* * * *

1 2 3 4
When we evaluate an automaton we can perform two kinds of transitions:

- When doing a “first child” move, we take black transitions (Down)
- When doing a “next sibling” move, we take red transitions (Right)
Initially, the stack contains the initial state
Example

```
startElement("c"), one Down transition, no Right transition
```
Example

```
{1},{}
{1},{}
{1},{}
```

```
startElement("d"), one Down transition, no Right transition
```
endElement("d"), replace last-sibling with the top of the stack
Example

startElement("a"), one Down transition, one Right transition
the Right transition goes to state 2, update the right of the stack
the Down transition goes to state 1, pushed on the stack
endElement("a")
From \{1\} \cup \{2\} compute the b transition
One Right (stay in state 2), replace right part of stack
Example

Two Down (stay in state 1, go to state 3), push on the stack
Example

```
c
ad b c b
d d d cc c
1 32
*
4a b d*
{1},{}
{1},{2}
{1,3},{}
{1,3},{}
```
Example

```
{1,3}, {}
{1,3}, {}
{1}, {2}
{1}, {}
```
Example

```
Example

d  a  b  c  b

  c  d  c  d  d  c

1  3  2

*  *

4  a  b  d*

{1},{}
{1},{2}
{1,3},{}
{1,3},{}
```

```
Example

{1,3,4},{}
{1,3},{}
{1,3},{}
{1,3,4},{}

```plaintext
1 3 2
* *
4 a b d*
```
Example
Example
Example

\[
\begin{array}{c}
\text{c} \\
\text{d a b c}
\end{array}
\]

{1,3},{}
{1,3},{}
{1,3},{}
{1},{}
Example

```
{1,3,4},{}
{1,3},{}
{1,3},{2}
{1},{}
```
Path with following-sibling and no filters

Adapt last week’s algorithm:

- keep a stack of pairs of sets of states
- the first set of states represents the states of the parent
- the second set of states represents the states of the previous-sibling
Path with following-sibling and no filters

Adapt last week’s algorithm:

- keep a stack of pairs of sets of states
- the first set of states represents the states of the parent
- the second set of states represents the states of the previous-sibling

More precisely, on `startElement`

1. Take the top of the stack $S_{parent}, S_{presib}$;
2. Compute the union $S = S_{parent} \cup S_{presib}$
3. Compute two new sets $S'_{parent}$ and $S'_{presib}$
4. $S'_{parent}$ is the set of states that can be reached from $S$ with a `Down` transition $S'_{presib}$ is the set of states that can be reached by a `Right` transition
5. At the top of the stack, replace $S_{presib}$ with $S'_{presib}$
6. Push $S'_{parent}, \{\}$ at the top of the stack
Adding preceding-sibling in filters

Consider `//a//b[./parent::c/preceding-sibling::d]/c`. What can we say about the b nodes?

- They must have a parent c
- The parent must have a preceding-sibling d

This is true for all the nodes which are:

- below a c
- which is a following sibling of a d
- which can occur anywhere

⇒ `//d/following-sibling::c/*`

Only need to adapt last week’s algorithm to following-sibling
Some more examples of rewriting of filters

[./ancestor::d/preceding-sibling::e/parent::f] becomes
//f/e/following-sibling::d//*
Some more examples of rewriting of filters

[./ancestor::d/preceding-sibling::e/parent::f]
becomes
//f/e/following-sibling::d/*

[./preceding-sibling::e/ancestor::d/preceding-sibling::f]
becomes
//f/following-sibling::d//e/following-sibling::*
Some more examples of rewriting of filters

\[./\text{ancestor::d/preceding-sibling::e/parent::f}\]
becomes
\//\text{f/e/following-sibling::d/\ast}\]

\[./\text{preceding-sibling::e/ancestor::d/preceding-sibling::f}\]
becomes
\//\text{f/following-sibling::d//e/following-sibling::\ast}\]

What is the general rule?
Rewriting backward filters

Let \([ f \]) be a filter with backward axes. We rewrite it into a path \(d\). \(d\) and \(f\) do *NOT* compute the same results but, for every node selected by \(d\), \([ f \]) is true. Let:

\[
f = ./a_0::t_0/a_1::t_1/\ldots/a_n::t_n
\]

where \(a_i \in \{\text{parent, ancestor, preceding-sibling}\}\) and \(n\) \(t_i\) is a label or \(*\).

\[
//t_n/\bar{a}_n::t_{n-1}/\ldots/\bar{a}_0::*
\]

where \(\bar{a}_i\) is the inverse axis of \(a_i\) (\(\text{parent}\) is the inverse of \(\text{child}\), \(\text{descendant}\) the inverse of \(\text{ancestor}\) and \(\text{preceding-sibling}\) the inverse of \(\text{following-sibling}\)).