XML and Databases
Efficient XPath evaluation

Kim.Nguyen@nicta.com.au

Week 7
Given an XPath query, how to return the selected set of nodes?

- NodeSet algorithm: very easy, very inefficient
XPath

Given an XPath query, how to return the selected set of nodes?

- **NodeSet algorithm**: very easy, very inefficient
- **Automata-based algorithm**: very efficient, not too hard ;-)
Given an XPath query, how to return the selected set of nodes?

- NodeSet algorithm: very easy, very inefficient
- Automata-based algorithm: very efficient, not too hard ;-)
Given an XPath query, how to return the selected set of nodes?

- **NodeSet algorithm:** very easy, very inefficient
- **Automata-based algorithm:** very efficient, not too hard ;-)

We assume that the XPath query has been parsed into a sequence:

\[
p ::= [(a_1, l_1, p_1); \ldots; (a_n, l_n, p_n)]
\]

\[
a ::= \text{child} | \text{descendant} | \ldots
\]

\[
l ::= \ast | \text{tagname} | \text{text}()
\]
XPath

Given an XPath query, how to return the selected set of nodes?

- **NodeSet algorithm:** very easy, very inefficient
- **Automata-based algorithm:** very efficient, not too hard ;-)

We assume that the XPath query has been parsed into a sequence:

\[
p ::= [(a_1, l_1, p_1); \ldots; (a_n, l_n, p_n)]
\]

\[
a ::= \text{child} | \text{descendant} | \ldots
\]

\[
l ::= * | \text{tagname} | \text{text}()
\]

All the \( p_i \) have the form:

\[
p_i ::= [(a_{i1}, l_{i1}, []); \ldots; (a_{in}, l_{in}, [])]
\]
**XPath**

Given an XPath query, how to return the selected set of nodes?

- **NodeSet algorithm:** very easy, very inefficient
- **Automata-based algorithm:** very efficient, not too hard ;-)

We assume that the XPath query has been parsed into a sequence:

\[
p := [(a_1, l_1, p_1); \ldots; (a_n, l_n, p_n)]
\]

\[
a := \text{child} | \text{descendant} | \ldots
\]

\[
l := \ast | \text{tagname} | \text{text}()
\]

All the \(p_i\) have the form:

\[
p_i := [(a_{i1}, l_{i1}, []); \ldots; (a_{in}, l_{in}, [])]
\]
Node Set Algorithm (1/6)

NodeSet eval(Path p, NodeSet nodes,bool all)

Given a set of nodes of nodes of a document apply the path p to the set of node and returns:

- All the nodes matching the query if all is true
- The first node matching the query if all is false

NodeSet eval_axis(Axis a, Label l, NodeSet nodes,bool all)

Given a set of nodes of nodes of a document returns the nodes in the axis a with label l

- if all is true, returns all the matching nodes.
- if all is false, returns the first matching node
Node Set Algorithm (2/6)

NodeSet eval(Path p, NodeSet nodes, bool all) {
    NodeSet r = nodes;
    // we apply the steps one after another
    for each (a, l, f) in p {
        // we select all the node matching the axis and label
        r = eval_axis(a, l, r, all);
        if (filter != []) {
            r' = Empty;
            for each n in r
                if (eval(f, { n }, false) != Empty)
                    r' = add(r', n);
            r = r';
        }
    }
    return r;
}
Node Set Algorithm (3/6)

NodeSet eval_axis(Axis a, Label l, NodeSet n, bool all) {
  switch (a) {
  child: return eval_child(l, n, all);
  descendant: return eval_descendant(l, n, all);
  //continue for all the axes
  ...
  }
}
Node Set Algorithm (4/6)

NodeSet eval_descendant(Label l, NodeSet n, bool all) {
    NodeSet r = Empty;
    for each t in n {
        for each tc in children(t) {
            if (label(tc) == l) {
                r = add(r, tc);
                if (!all) // we only want the first result
                    return r;
            }
        } //r contains all the children of t tagged l
    } //r contains all the children of t tagged l
    r = r ∪ eval_descendant(l, children(t));
}

return r;
Node Set Algorithm (5/6)

Example: XPath expression //a[d//e]/b//c
Called initially with the NodeSet containing the root

```
[6x258]No de Set Algo rithm (5/6)
[6x223]Example: XP ath exp resison //a[d//e]/b//c
[6x208]Called initially with the NodeSet containing the root
```
Example: XPath expression //a[d//e]/b//c
Called initially with the NodeSet containing the root
Example: XPath expresison //a[d//e]/b//c
Called initially with the NodeSet containing the root

eval_axis(desc,a,...,true)
Example: XPath expression //a[d//e]/b//c
Called initially with the NodeSet containing the **root**

```
Result of the first step
```

```
eval_axis(desc,a,...,true)
```

```
eval(d//e,...,false)
```

```
Result of the second step
```

```
Final result
```

Example: XPath expression //a[d//e]/b//c
Called initially with the NodeSet containing the root

```
 eval_axis(desc,a,..,true)
eval(d//e,..,false)
Result of the first step
```
Example: XPath expression //a[d/e]/b/c
Called initially with the NodeSet containing the root

```
eval_axis(desc,a,...,true)
eval(d/e,...,false)
```
Result of the first step
```
eval_axis(child,b,...,true)
```
Final result
Example: XPath expression //a[d//e]/b//c
Called initially with the NodeSet containing the root

eval_axis(desc,a,\ldots,true)
eval(d//e,\ldots,false)
Result of the first step
eval_axis(child,b,\ldots,true)
Result of the 2\textsuperscript{nd} step
Example: XPath expression //a[//d//e]///b////c
Called initially with the NodeSet containing the root

eval_axis(desc,a,\ldots,\text{true})
eval(d//e,\ldots,\text{false})
Result of the first step
eval_axis(child,b,\ldots,\text{true})
Result of the 2nd step
eval_axis(desc,c,\ldots,\text{true})
Example: XPath expression //a[d//e]/b//c
Called initially with the NodeSet containing the root

eval_axis(desc,a,...,true)
eval(d//e,...,false)
Result of the first step
eval_axis(child,b,...,true)
Result of the 2nd step
eval_axis(desc,c,...,true)
Final result
Pros and cons of the algorithm:

+ Easy to implement

Remains very inefficient: \( O(|D|^2) \) for forward XPath, \( O(2^{|Q|} + |D|^2) \) for full XPath (cf. Lecture)
Pros and cons of the algorithm:

- Easy to implement
- Can can be extended to all XPath axes easily

Remains very inefficient: $O(|D|^2)$ for forward XPath, $O(2^{|Q|} + |D|^2)$ for full XPath (cf. Lecture)
Pros and cons of the algorithm:

+ Easy to implement
+ Can can be extended to all XPath axes easily
  - May return several copies of the same node, thus either use a Set datastructure for the result, or sort and sieve the result at the end.

Remains very inefficient: $O(|D|^2)$ for forward XPath, $O(2^{|Q|} + |D|^2)$ for full XPath (cf. Lecture)
Pros and cons of the algorithm:

+ Easy to implement
+ Can be extended to all XPath axes easily
  - May return several copies of the same node, thus either use a Set datastructure for the result, or sort and sieve the result at the end.
  - Need to traverse many times the tree, cannot be done in streaming

Remains very inefficient: $O(|D|^2)$ for forward XPath, $O(2^{|Q|} + |D|^2)$ for full XPath (cf. Lecture)
Automata based algorithm

We proceed in two steps:

- first we see how this works for XPath expressions without filters
Automata based algorithm

We proceed in two steps:

- first we see how this works for XPathVariable expressions without filters
- we add filters
Automata based algorithm

We proceed in two steps:

- first we see how this works for XPath expressions without filters
- we add filters
Automata based algorithm

We proceed in two steps:
  1. first we see how this works for XPath expressions without filters
  2. we add filters

The idea is to see the XPath expression as a regular expression matching the paths of the tree. The translation of a forward XPath expression into an NFA is straightforward:
Automata based algorithm

We proceed in two steps:

- first we see how this works for XPath expressions without filters
- we add filters

The idea is to see the XPath expression as a regular expression matching the paths of the tree. The translation of a forward XPath expression into an NFA is straightforward:

```
//a/b//c  becomes:
```

```
Automata based algorithm

We proceed in two steps:

▶ first we see how this works for XPath expressions without filters
▶ we add filters

The idea is to see the XPath expression as a regular expression matching the paths of the tree. The translation of a forward XPath expression into an NFA is straightforward:

```
//a/b//c becomes: 1 → 2 → 3 → 4
```

If we determinise, it becomes:

```
1, 2
1, 3
1, 3, 4
1, 2, 3
```
Automata based algorithm

We proceed in two steps:

- first we see how this works for XPath expressions without filters
- we add filters

The idea is to see the XPath expression as a regular expression matching the paths of the tree. The translation of a forward XPath expression into an NFA is straightforward:

//a//b//c becomes:  

If we determinise, it becomes:
Automata based algorithm

We proceed in two steps:
- first we see how this works for XPath expressions without filters
- we add filters

The idea is to see the XPath expression as a regular expression matching the paths of the tree. The translation of a forward XPath expression into an NFA is straightforward:

//a/b//c becomes:

If we determinise, it becomes:
Problems of determinisation:

- Exponential blow-up in the number of states
Automata based algorithm

Problems of determinisation:

- Exponential blow-up in the number of states
- computing the *default* transition * is tricky!
Automata based algorithm

Problems of determinisation:

- Exponential blow-up in the number of states
- computing the default transition * is tricky!
Automata based algorithm

Problems of determinisation:

- Exponential blow-up in the number of states
- computing the default transition * is tricky!

Good news: we don’t need to determinize!

Reference:

*Processing XML streams with deterministic automata and stream indexes*
By T.J. Green, A. Gupta, G. Miklau, M. Onizuka, D. Suciu, TODS 2004
Topdown XPath evaluation

// Takes a NFA, a set of states and a document node
// Returns the set of nodes matched by the automaton
void evaluate(NodeSet eval(Automaton a, States S, Node t) { // The empty tree yields no result
    if (t == null) return Empty;
    else { // Everything is done here, see next slide
        S' = {q' | ∀q ∈ S, s.t. q, l → q' ∈ a, l = label(t) or *}
        r = Empty;
        for each t' in children(t) {
            r = r UNION eval(a, S', t');
        }
        if (finalstate(a) ∈ S')
            r = r UNION {t}
    }
    return r;
}
Topdown XPath evaluation

What does this do?

\[ S' = \{ q' \mid \forall q \in S, \text{s.t. } q, l \rightarrow q' \in a, \ l = \text{label}(t) \text{ or } * \} \]
Topdown XPath evaluation

What does this do?

\[ S' = \{ q' \mid \forall q \in S, \text{ s.t. } q, l \rightarrow q' \in a, \text{ } l = \text{label}(t) \text{ or } * \}\]

For each state \( q \) of the NFA in \( S \) it computes the set of states in which we can go with the label of the current node \( t \)

- Then we recursively evaluate \( S' \) on all the children of \( t \)
- If we took a transition which lead us to an accept state, then we also need to add \( t \) to the final result
Topdown XPath evaluation

What does this do?

\[ S' = \{ q' \mid \forall q \in S, \text{s.t. } q, l \rightarrow q' \in a, \ l = \text{label}(t) \text{ or } * \} \]

For each state \( q \) of the NFA in \( S \) it computes the set of states in which we can go with the label of the current node \( t \)

- Then we recursively evaluate \( S' \) on all the children of \( t \)
- If we took a transition which lead us to an accept state, then we also need to add \( t \) to the final result

To represent the NFA, we need:

- The set of all states, \( Q \), the initial state \( I \), the final state \( F \)
- a hash table mapping pairs of states \( \times \) labels to states
Topdown XPath evaluation

\[ Q = \{1, 2, 3, 4\} \]
\[ I = \{1\} \]
\[ F = \{4\} \]

1. \( a \mapsto 2 \)
2. \( 1, \ast \mapsto 1 \)
3. \( b \mapsto 3 \)
4. \( 3, c \mapsto 4 \)
5. \( 3, c \mapsto 3 \)
Topdown XPath evaluation

We start on the root, with the initial state

\[ Q = \{1, 2, 3, 4\} \]
\[ I = \{1\} \]
\[ F = \{4\} \]

1, a \mapsto 2
1, * \mapsto 1
2, b \mapsto 3
3, c \mapsto 4
3, c \mapsto 3
Topdown XPath evaluation

For label “a” in state 1, the NFA can end up in two states, 1 and 2...

\[
Q = \{1, 2, 3, 4\} \\
I = \{1\} \\
F = \{4\}
\]

\[
\begin{align*}
1, a & \mapsto 2 \\
1, * & \mapsto 1 \\
2, b & \mapsto 3 \\
3, c & \mapsto 4 \\
3, c & \mapsto 3
\end{align*}
\]
Topdown XPath evaluation

So we call recursively, with $S = \{1, 2\}$ on the first child of the root...

$Q = \{1, 2, 3, 4\}$
$I = \{1\}$
$F = \{4\}$

1. a $\mapsto$ 2
2. * $\mapsto$ 1
3. b $\mapsto$ 3
3. c $\mapsto$ 4
3. c $\mapsto$ 3
Topdown XPath evaluation

\[
Q = \{1, 2, 3, 4\}
\]
\[
I = \{1\}
\]
\[
F = \{4\}
\]

\[
\begin{align*}
1, a & \mapsto 2 \\
1, * & \mapsto 1 \\
2, b & \mapsto 3 \\
3, c & \mapsto 4 \\
3, c & \mapsto 3
\end{align*}
\]
Topdown XPath evaluation

\[ Q = \{1, 2, 3, 4\} \]
\[ I = \{1\} \]
\[ F = \{4\} \]

1. a \mapsto 2
2. * \mapsto 1
3. b \mapsto 3
4. c \mapsto 4
5. c \mapsto 3
Topdown XPath evaluation

Here label “b” allows us to go in state 3 and also stays in state 1

\[ Q = \{1, 2, 3, 4\} \]
\[ I = \{1\} \]
\[ F = \{4\} \]

1, a ↦ 2
1, * ↦ 1
2, b ↦ 3
3, c ↦ 4
3, c ↦ 3
Topdown XPath evaluation

Q = \{1, 2, 3, 4\}
I = \{1\}
F = \{4\}

1, a \mapsto 2
1, * \mapsto 1
2, b \mapsto 3
3, c \mapsto 4
3, c \mapsto 3
Topdown XPath evaluation

We arrive in “c”. The call on the children returns Empty. One of our state is final, so there is a run of the automaton which accepts this path, we mark the node as selected.

Q = \{1, 2, 3, 4\}
I = \{1\}
F = \{4\}

1, a ↦→ 2
1, * ↦→ 1
2, b ↦→ 3
3, c ↦→ 4
3, c ↦→ 3
Topdown XPath evaluation

\[ Q = \{1, 2, 3, 4\} \]
\[ I = \{1\} \]
\[ F = \{4\} \]

1, a \mapsto 2
1, * \mapsto 1
2, b \mapsto 3
3, c \mapsto 4
3, c \mapsto 3
Topdown XPath evaluation

\[
Q = \{1, 2, 3, 4\} \\
I = \{1\} \\
F = \{4\} \\
1, a \mapsto 2 \\
1, * \mapsto 1 \\
2, b \mapsto 3 \\
3, c \mapsto 4 \\
3, c \mapsto 3
\]
Topdown XPath evaluation

Q = \{1, 2, 3, 4\}
I = \{1\}
F = \{4\}

1, a \mapsto 2
1, * \mapsto 1
2, b \mapsto 3
3, c \mapsto 4
3, c \mapsto 3
Topdown XPath evaluation

\begin{itemize}
  \item $Q = \{1, 2, 3, 4\}$
  \item $I = \{1\}$
  \item $F = \{4\}$
\end{itemize}
Topdown XPath evaluation

\[
Q = \{1, 2, 3, 4\}
\]
\[
I = \{1\}
\]
\[
F = \{4\}
\]

1, a \mapsto 2
1, * \mapsto 1
2, b \mapsto 3
3, c \mapsto 4
3, c \mapsto 3
Topdown XPath evaluation

\[
Q = \{1, 2, 3, 4\}\\
I = \{1\}\\
F = \{4\}\\
\]

1, a \mapsto 2\\
1, * \mapsto 1\\
2, b \mapsto 3\\
3, c \mapsto 4\\
3, c \mapsto 3
Topdown XPath evaluation

\[ Q = \{1, 2, 3, 4\} \]
\[ I = \{1\} \]
\[ F = \{4\} \]

1, a ↦ 2
1, * ↦ 1
2, b ↦ 3
3, c ↦ 4
3, c ↦ 3
Topdown XPath evaluation

\[ Q = \{1, 2, 3, 4\} \]
\[ I = \{1\} \]
\[ F = \{4\} \]

1, a \mapsto 2
1, * \mapsto 1
2, b \mapsto 3
3, c \mapsto 4
3, c \mapsto 3
Topdown XPath evaluation

\[ Q = \{1, 2, 3, 4\} \]
\[ I = \{1\} \]
\[ F = \{4\} \]

1, a \mapsto 2
1, * \mapsto 1
2, b \mapsto 3
3, c \mapsto 4
3, c \mapsto 3
Topdown XPath evaluation

\[ Q = \{1, 2, 3, 4\} \]

\[ I = \{1\} \]

\[ F = \{4\} \]
Topdown XPath evaluation

Q = \{1, 2, 3, 4\}
I = \{1\}
F = \{4\}

1, a \mapsto 2
1, * \mapsto 1
2, b \mapsto 3
3, c \mapsto 4
3, c \mapsto 3
Topdown XPath evaluation

$Q = \{1, 2, 3, 4\}$

$I = \{1\}$

$F = \{4\}$

1, a $\mapsto$ 2
1, * $\mapsto$ 1
2, b $\mapsto$ 3
3, c $\mapsto$ 4
3, c $\mapsto$ 3
Topdown XPath evaluation

Q = \{1, 2, 3, 4\}
I = \{1\}
F = \{4\}

1, a \mapsto 2
1, * \mapsto 1
2, b \mapsto 3
3, c \mapsto 4
3, c \mapsto 3
Topdown XPath evaluation

\[ Q = \{1, 2, 3, 4\} \]
\[ I = \{1\} \]
\[ F = \{4\} \]

1, a ⟷ 2
1, * ⟷ 1
2, b ⟷ 3
3, c ⟷ 4
3, c ⟷ 3
Topdown XPath evaluation

\[ Q = \{1, 2, 3, 4\} \]
\[ I = \{1\} \]
\[ F = \{4\} \]

1, a \mapsto 2
1, * \mapsto 1
2, b \mapsto 3
3, c \mapsto 4
3, c \mapsto 3
Topdown XPath evaluation

- $Q = \{1, 2, 3, 4\}$
- $I = \{1\}$
- $F = \{4\}$

- 1, a $\mapsto$ 2
- 1, * $\mapsto$ 1
- 2, b $\mapsto$ 3
- 3, c $\mapsto$ 4
- 3, c $\mapsto$ 3
Topdown XPathVariable evaluation

Q = \{1, 2, 3, 4\}
I = \{1\}
F = \{4\}

\[
\begin{align*}
1, a & \mapsto 2 \\
1, * & \mapsto 1 \\
2, b & \mapsto 3 \\
3, c & \mapsto 4 \\
3, c & \mapsto 3
\end{align*}
\]
Topdown XPath evaluation

\[ Q = \{ 1, 2, 3, 4 \} \]
\[ I = \{ 1 \} \]
\[ F = \{ 4 \} \]

1, a $\mapsto$ 2
1, * $\mapsto$ 1
2, b $\mapsto$ 3
3, c $\mapsto$ 4
3, c $\mapsto$ 3
Topdown XPath evaluation

What is the complexity of the algorithm?

- We do only one pre-order traversal

\[ S' = \{ q' | \forall q \in S, \text{s.t. } q, l \rightarrow q' \in a, l = \text{label}(t) \text{ or } * \} \Rightarrow \text{this is linear in the size of } S, \text{which is at most as big as the number of states in the NFA. As we have seen, the number of states is linear in the size of the query so this operation costs } |Q| \]

\[ r = r \cup \{ t \} \Rightarrow \text{Since we traverse the tree in pre-order and only once for every node we can use a list for the result set, and just add } \{ t \} \text{ at the beginning, which is constant time. In particular, we don't have to sort the result, nor use a data structure with } |O(\log(n))| \text{ insertion to guarantee the right order nor do we have to filter the results to remove duplicates: huge improvement.} \]

Complexity is combined linear time \( O(|Q| \times |D|) \), which is the best complexity for this problem (cf lecture).
Topdown XPath evaluation

What is the complexity of the algorithm?

- We do only one pre-order traversal
- For each node, we perform the following:

\[
\begin{align*}
S' &= \{ q' \mid \forall q \in S, \text{s.t. } q, l \rightarrow q' \in a, l = \text{label}(t) \text{ or } * \} \\
\Rightarrow & \text{this is linear in the size of } S, \text{ which is at most as big as the number of states in the NFA. As we have seen, the number of states is linear in the size of the query so this operation costs } |Q| \\
r &= r \cup \{ t \} \\
\Rightarrow & \text{Since we traverse the tree in pre-order and only once for every node we can use a list for the result set, and just add } \{ t \} \text{ at the beginning, which is constant time. In particular, we don't have to sort the result, nor use a data structure with } |O(\log(n))| \text{ insertion to guarantee the right order nor do we have to filter the results to remove duplicates: huge improvement.}
\end{align*}
\]

Complexity is combined linear time $O(|Q| \times |D|)$, which is the best complexity for this problem (cf lecture).
Topdown XPath evaluation

What is the complexity of the algorithm?

- We do only one pre-order traversal
- For each node, we perform the following:

  1. $S' = \{ q' \mid \forall q \in S, \text{s.t. } q, l \rightarrow q' \in a, \ l = \text{label}(t) \text{ or } * \}$
Topdown XPath evaluation

What is the complexity of the algorithm?

- We do only one pre-order traversal
- For each node, we perform the following:

  1. \[ S' = \{ q' | \forall q \in S, \text{s.t. } q, l \rightarrow q' \in a, \text{ } l = \text{label}(t) \text{ or } * \}\]

  \[ \Rightarrow \text{this is linear in the size of } S, \text{ which is at most as big as the number of states in the NFA. As we have seen, the number of states is linear in the size of the query so this operation costs } |Q| \]
Topdown XPath evaluation

What is the complexity of the algorithm?

- We do only one pre-order traversal
- For each node, we perform the following:
  1. \( S' = \{ q' \mid \forall q \in S, \text{s.t.} \ q, l \rightarrow q' \in a, \ l = \text{label}(t) \text{ or } * \} \)
     \( \Rightarrow \) this is linear in the size of \( S \), which is at most as big as the number of states in the NFA. As we have seen, the number of states is linear in the size of the query so this operation costs \(|Q|\)
  2. \( r = r \cup \{ t \} \)
Topdown XPath evaluation

What is the complexity of the algorithm?

▶ We do only one pre-order traversal
▶ For each node, we perform the following:

1. \[ S' = \{ q' \mid \forall q \in S, \text{s.t. } q, l \rightarrow q' \in a, l = \text{label}(t) \text{ or } * \} \]
   \[ \Rightarrow \] this is linear in the size of \( S \), which is at most as big as the number of states in the NFA. As we have seen, the number of states is linear in the size of the query so this operation costs \( |Q| \)

2. \[ r = r \cup \{ t \} \]
   \[ \Rightarrow \] Since we traverse the tree in pre-order and only once for every node we can use a list for the result set, and just add \( \{ t \} \) at the beginning, which is constant time. In particular, we don’t have to sort the result, nor use a data structure with \( |O(\log(n))| \) insertion to guarantee the right order nor do we have to filter the results to remove duplicates: huge improvement.

Complexity is combined linear time \( O(|Q| \times |D|) \), which is the best complexity for this problem (cf lecture).
Topdown XPath evaluation

What is the complexity of the algorithm?

- We do only one pre-order traversal
- For each node, we perform the following:
  1. $S' = \{q' \mid \forall q \in S, \text{s.t. } q, l \rightarrow q' \in a, l = \text{label}(t) \text{ or } *\}$
     $\Rightarrow$ this is linear in the size of $S$, which is at most as big as the number of states in the NFA. As we have seen, the number of states is linear in the size of the query so this operation costs $|Q|$
  2. $r = r \cup \{t\}$
     $\Rightarrow$ Since we traverse the tree in pre-order and only once for every node we can use a list for the result set, and just add $\{t\}$ at the beginning, which is constant time. In particular, we don’t have to sort the result, nor use a data structure with $|O(\log(n))|$ insertion to guarantee the right order nor do we have to filter the results to remove duplicates: huge improvement.

Complexity is combined linear time $O(|Q| \times |D|)$, which is the best complexity for this problem (cf lecture).
Topdown XPath evaluation

What is the complexity of the algorithm?

- We do only one pre-order traversal
- For each node, we perform the following:
  1. $S' = \{ q' \mid \forall q \in S, \text{s.t. } q, l \rightarrow q' \in a, \text{ } l = \text{label}(t) \text{ or } * \}$
     $\Rightarrow$ this is linear in the size of $S$, which is at most as big as the number of states in the NFA. As we have seen, the number of states is linear in the size of the query so this operation costs $|Q|$
  2. $r = r \cup \{ t \}$
     $\Rightarrow$ Since we traverse the tree in pre-order and only once for every node we can use a list for the result set, and just add $\{ t \}$ at the beginning, which is constant time. In particular, we don’t have to sort the result, nor use a data structure with $O(\log(n))$ insertion to guarantee the right order nor do we have to filter the results to remove duplicates: huge improvement.

Complexity is combined linear time $O(|Q| \times |D|)$, which is the best complexity for this problem (cf lecture).
Topdown XPath evaluation

How do we add filters?
Topdown XPath evaluation

How do we add filters?
Consider:

```
//a[d//e]/b//c
```
Topdown XPath evaluation

How do we add filters?
Consider:

\[ //a[ d//e ]/b//c \]

We build two automata:
Topdown XPath evaluation

How do we add filters?
Consider:

```
//a[ d//e ]/b//c
```

We build two automata:
Topdown XPath evaluation

NodeSet eval(Automata a, States S, Node t, FilterStack FS)
{
    if (t == null) return Empty, FS
    else {
        S' = \{ q' | q, l → q' ∈ a, l = label(t) or * \}
        FilterSet f = \{ \{ InitState(FilterAuto(q)) \} | q ∈ S \}
        FS' = push(f, FS);
        FS'' = EmptyStack;
        for each fs in FS' {
            fs' = Empty;
            for each (_, s) in fs
                fs' = fs' ∪ \{ s × \{ q' | q, l → q' ∈ a_i, l = label(t) or * \} \}
            push(FS'', fs');
        }
    }
    ...
Topdown XPath evaluation

\[
\begin{align*}
    r &= \text{Empty} \\
    fs &= \text{Empty} \\
    \text{for each } t' \text{ in children}(t) \{ \\
        r',FS''' &= \text{eval}(a,S',t',FS''') \\
        r &= r \cup r' \\
        fs''' &= \text{pop}(FS''') \\
        fs'' &= \text{pop}(FS'') \\
        \text{for each } (s,s') \text{ in } fs'' \{ \\
            \text{if } (\text{finalstate}(a')_{\cup} \in s') \\
            &\quad \text{remove } (_,s) \text{ from } fs''; \\
            FS'' &= \text{push}(FS'',fs'') \\
        \}
    \}
\end{align*}
\]
Topdown XPath evaluation

```
fs = peek(FS’’);
if (isempty(fs))
    if (finalstate(a) ∈ S)
        r = r ∪ {t};
else
    r = Empty
return (r,FS’’);
```
For the assignment?

- DOM, Dag: that’s the same, you only need to implement FirstChild/NextSibling correctly for the Dag!
For the assignment?

- DOM, Dag: that’s the same, you only need to implement FirstChild/NextSibling correctly for the Dag!
- SAX: You rewrite the recursive function eval into SAX events, and manage a stack yourself, but the principle is the same.
For the assignment?

- DOM, Dag: that’s the same, you only need to implement FirstChild/NextSibling correctly for the Dag!
- SAX: You rewrite the recursive function eval into to SAX events, and manage a stack yourself, but the principle is the same.
- Text nodes? You can put a special label $ in the automata and use it to match PCDATA_NODE DOM nodes or characters() SAX events
For the assignment?

- DOM, Dag: that’s the same, you only need to implement FirstChild/NextSibling correctly for the Dag!
- SAX: You rewrite the recursive function eval into SAX events, and manage a stack yourself, but the principle is the same.
- Text nodes? You can put a special label $ in the automata and use it to match PCDATA_NODE DOM nodes or characters() SAX events
- following-siblings? You keep the following sibling transitions in your automaton, and trigger them during the children iteration
For the assignment?

- DOM, Dag: that’s the same, you only need to implement FirstChild/NextSibling correctly for the Dag!
- SAX: You rewrite the recursive function eval into to SAX events, and manage a stack yourself, but the principle is the same.
- Text nodes? You can put a special label $ in the automata and use it to match PCDATA_NODE DOM nodes or characters() SAX events
- following-siblings? You keep the following sibling transitions in your automaton, and trigger them during the children iteration
For the assignment?

- DOM, Dag: that’s the same, you only need to implement FirstChild/NextSibling correctly for the Dag!

- SAX: You rewrite the recursive function eval into to SAX events, and manage a stack yourself, but the principle is the same.

- Text nodes? You can put a special label $ in the automata and use it to match PCDATA_NODE DOM nodes or characters() SAX events

- following-siblings? You keep the following sibling transitions in your automaton, and trigger them during the children iteration