Deterministic Regular Expressions in Linear Time

reasoning about deterministic regular expressions without building the automata

Benoit Groz[1,3], Sebastian Maneth[2,3], Slawek Staworko[1,3]

[1]University of Lille, Mostrare INRIA

[2]UNSW, NICTA

[3]Équipe associée Transduce

PODS, May, 2012

INRIA centre de recherche LILLE - NORD EUROPE
Université Lille1 Sciences et Technologies
NICTA
UNSW THE UNIVERSITY OF NEW SOUTH WALES

Context

DTD and XML Schema use regular expressions to specify which sequence of children may appear below a node.

```
<!ELEMENT book (author,chapter*,index?)>
```

Constraint: regular expression must be deterministic.

We provide new algorithms to:

- Check if a regular expression is deterministic.
- Decide the membership problem for deterministic regular expressions.

Outline

Outline

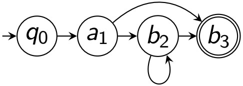
Structure of regular expressions

ab^*b

abb^*

Structure of regular expressions

$a_1 b_2^* b_3$

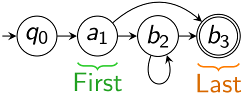


b_3 *follows* a_1, b_2 *follows* $a_1 \ldots$

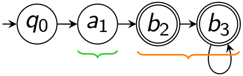
$a_1 b_2 b_3^*$

Structure of regular expressions

$a_1 b_2^* b_3$

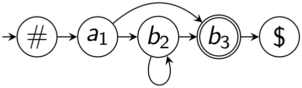


$a_1 b_2 b_3^*$

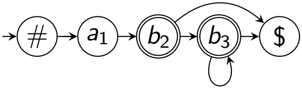


Structure of regular expressions

$\# a_1 b_2^* b_3 \$$



$\# a_1 b_2 b_3^* \$$

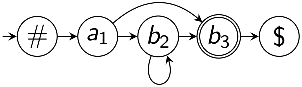


Deterministic regular expressions (a.k.a. one-unambiguous)

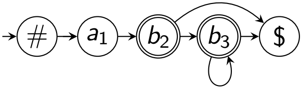
Expression is *non deterministic* if:

$$b_i \longrightarrow a_j \qquad a_k \qquad (j \neq k)$$

$\#a_1 b_2^* b_3 \$$
$\Rightarrow$ *non deterministic*



$\#a_1 b_2 b_3^* \$$
$\Rightarrow$ *deterministic*

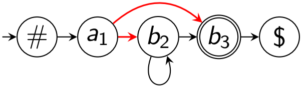


Deterministic regular expressions (a.k.a. one-unambiguous)

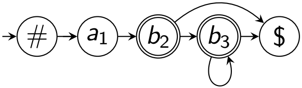
Expression is *non deterministic* if:

$$\begin{tikzcd} b_i \arrow[r, bend left] \arrow[rr, bend left] & a_j & a_k & (j \neq k) \end{tikzcd}$$

$\# a_1 b_2^* b_3 \$$
$\Rightarrow$ *non deterministic*



$\# a_1 b_2 b_3^* \$$
$\Rightarrow$ *deterministic*

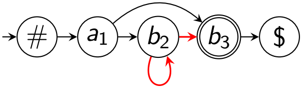


Deterministic regular expressions (a.k.a. one-unambiguous)

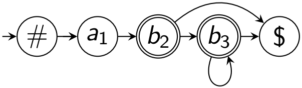
Expression is *non deterministic* if:

$$\begin{tikzcd} b_i \arrow[r, bend left] \arrow[rr, bend left] & a_j & a_k \end{tikzcd} \qquad (j \neq k)$$

$\# a_1 b_2^* b_3 \$$
$\Rightarrow$ *non deterministic*



$\# a_1 b_2 b_3^* \$$
$\Rightarrow$ *deterministic*

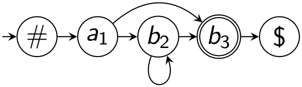


Deterministic regular expressions (a.k.a. one-unambiguous)

Expression is *non deterministic* if:

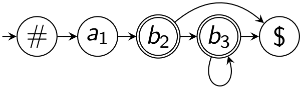
$$b_i \qquad a_j \qquad a_k \qquad (j \neq k)$$

$\#a_1 b_2^* b_3\$$
$\Rightarrow$*non deterministic*



Ambiguity parsing $w = ab$

$\#a_1 b_2 b_3^*\$$
$\Rightarrow$*deterministic*

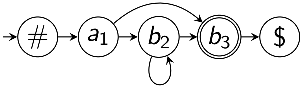


Deterministic regular expressions (a.k.a. one-unambiguous)

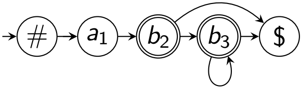
Expression is *non deterministic* if:

$$b_i \qquad a_j \qquad a_k \qquad (j \neq k)$$

$\#a_1 b_2^* b_3 \$$
$\Rightarrow$ *non deterministic*



$\#a_1 b_2 b_3^* \$$
$\Rightarrow$ *deterministic*



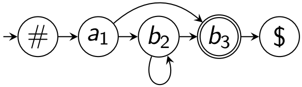
$e = (a+b)b?(ab)^* \quad ?$
$e = (ab+ba?)^* \quad ?$

Deterministic regular expressions (a.k.a. one-unambiguous)

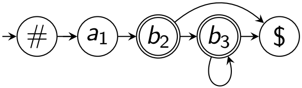
Expression is *non deterministic* if:

$$b_i \qquad a_j \qquad a_k \qquad (j \neq k)$$

$\# a_1 b_2^* b_3 \$$
$\Rightarrow$ *non deterministic*



$\# a_1 b_2 b_3^* \$$
$\Rightarrow$ *deterministic*



$e = (a + b)b?(ab)^*$ $\quad\Rightarrow$ *deterministic*
$e = (ab+ba?)^*$ $\quad\Rightarrow$ *non deterministic:* $w = ba$

Outline

Problems of interest

Testing determinism:

Input: expression e,
Question: is e deterministic?

Membership:

Input: word w, deterministic expression e,
Question: $w \in L(e)$?

Scenario: big expression, big alphabet.

Remark:

size of e $=$ number of nodes in the parse tree
$\simeq$ number of positions.

Problems of interest

Testing determinism:

Input: expression e,
Question: is e deterministic?

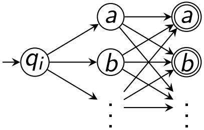
Membership:

Input: word w, deterministic expression e,
Question: $w \in L(e)$?

Straightforward solution through Glushkov automaton.
Build Glushkov in $O(|\Sigma| \times |e|)$ [Brüggeman-Klein TCS'93].
$\Longrightarrow$ (quadratic in $|e|$)

Number of transitions of Glushkov can be quadratic:

$e = (a + b + c \ldots)(a + b + c \ldots),$
$e' = (a + b + c \ldots)^*,$
$e'' = (a?b?c? \ldots)$



Problems of interest

Testing determinism:

Input: expression e,
Question: is e deterministic?

Membership:

Input: word w, deterministic expression e,
Question: $w \in L(e)$?

Straightforward solution through Glushkov automaton.
Build Glushkov in $O(|\Sigma| \times |e|)$ [Brüggeman-Klein TCS'93].
$\Longrightarrow$ (quadratic in $|e|$)

Number of transitions of Glushkov can be quadratic:

$e = (a + b + c \ldots)(a + b + c \ldots)$,
$e' = (a + b + c \ldots)^*$,
$e'' = (a?b?c? \ldots)$

Problems of interest

Testing determinism:

Input: expression e,
Question: is e deterministic?

Membership:

Input: word w, deterministic expression e,
Question: $w \in L(e)$?

Straightforward solution through Glushkov automaton.
Build Glushkov in $O(|\Sigma| \times |e|)$ [Brüggeman-Klein TCS'93].
$\Longrightarrow$ (quadratic in $|e|$)

Can we do better?

Summary

	Glushkov	Our results
testing determinism	$O(\|\Sigma\| \times \|e\|)$	$O(\|e\|)$
membership	$O(\|\Sigma\| \times \|e\| + \|w\|)$	$O(\|e\| + \|w\| \log\log(e))$
⋆ k-occurrence		$O(k \times \|w\|)$
⋆ restrictions on $+$		$O(\|e\| + \|w\|)$
⋆ star free		$O(\|e\| + \|w\|)$

This talk

Roadmap

Preprocessing

We do not construct automaton. Instead we work on parse tree which we preprocess in linear time to build some pointers+datastructures.

- **Testing determinism.**
  We search a witness for non-determinism in e: pair of two positions with same label that follow a common position.

$$\begin{tikzcd} b_i \arrow[r] \arrow[rr, bend left] & a_j & a_k \end{tikzcd} \qquad (j \neq k)$$

We limit the number of pairs examined to $O(|e|)$ using skeleta from [Bojańczyk and Parys JACM'11].

- **Testing membership.**
  We simulate transitions on-the-fly using the pointers from preprocessing.

Roadmap: transition simulation

We want a procedure for transition simulation:

Input:

a position a_i in the expression,

a letter b (the next letter of the word)

Output:

the unique b-labeled position that follows a_i

Outline

Some tools

LCA preprocessing [Harel&Tarjan, SICOMP'84]

One can preprocess a tree in linear time, to answer lowest common ancestor (LCA) queries and ancestor queries in constant time.

(Input for preprocessing: t)

LCA query:

Input: two nodes n_1, n_2 in t

Output: the lowest common ancestor (LCA) of n_1 and n_2

Ancestor query:

Input: two nodes n_1, n_2 in t

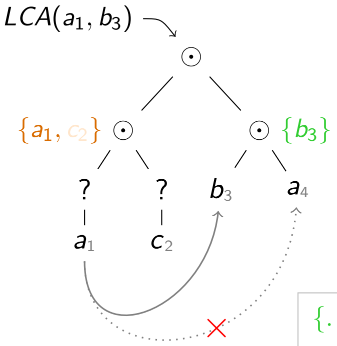
Question: is n_1 an ancestor of n_2?

Work on parse tree

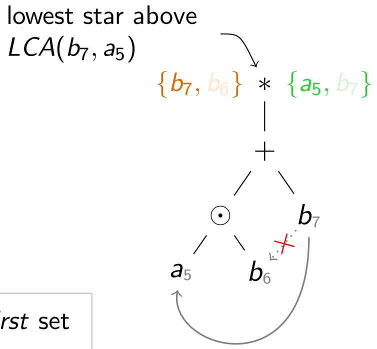
Goal: Given two positions a_i and b_j,
test if b_j follows a_i in constant time.

Work on parse tree: Follow

When does b_j follow a_i?



Case ❶ : $\odot$



Case ❷ : $*$

$\{\dots\}$	:	*First* set
$\{\dots\}$	:	*Last* set

Work on parse tree: Follow (2)

Assuming that after some preprocessing we can test *First* and *Last* in constant time,

Theorem

We can test if b_j follows a_i in constant time.

Work on parse tree: Follow (2)

Assuming that after some preprocessing we can test *First* and *Last* in constant time,

Theorem

We can test if b_j follows a_i in constant time.

Preprocessing:

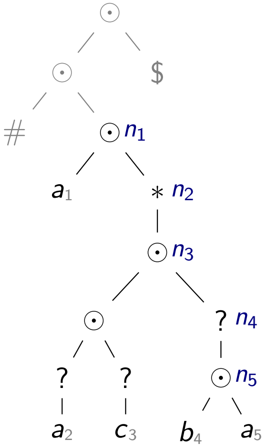
pointer to lowest $*$ ancestor of each node.
build LCA, *First* and *Last* structures.

Algo. to test if b_j follows a_i:

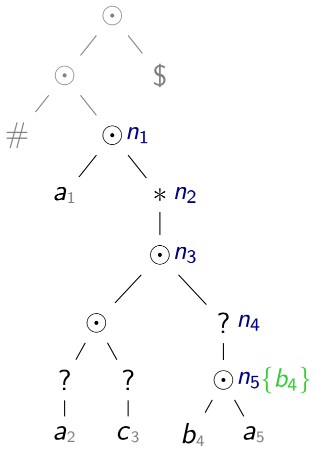
compute $LCA(b_j, a_i)$
follow the $*$ pointer (for case 2: $*$)
test that a_i, b_j in *First* and *Last* of appropriate nodes

New objective: test if $a_i \in First(n)$ in constant time

Work on parse tree: First and Last

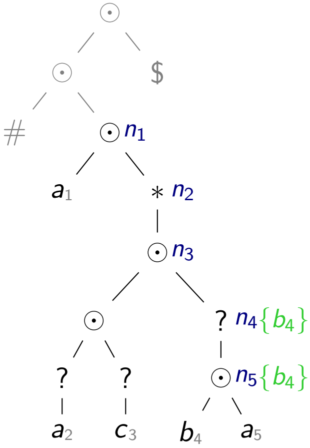


Work on parse tree: First and Last



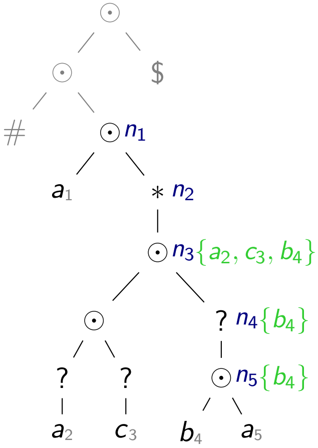
$\{a_2, \ldots\}$: *First* set

Work on parse tree: First and Last



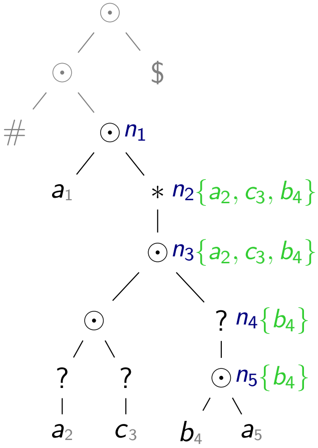
$\{a_2, \dots\}$: *First* set

Work on parse tree: First and Last



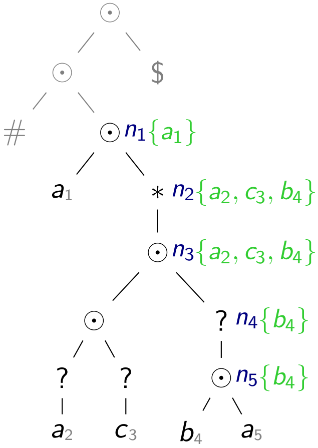
$\{a_2, \ldots\}$: *First* set

Work on parse tree: First and Last



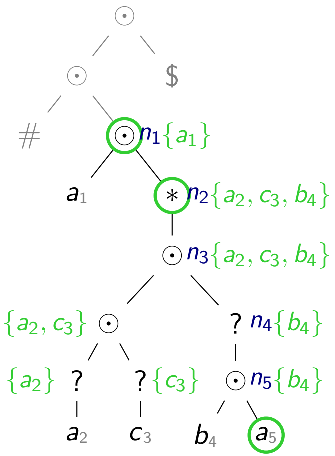
$\{a_2, \ldots\}$: *First* set

Work on parse tree: First and Last



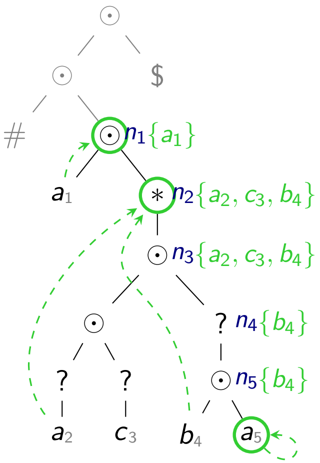
$\{a_2, \ldots\}$: *First* set

Work on parse tree: First and Last



$\{a_2, \ldots\}$: *First* set

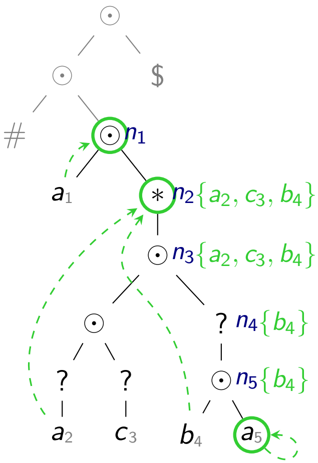
Work on parse tree: First and Last



$\{a_2, \ldots\}$: *First* set

: a *SupFirst* pointer

Work on parse tree: First and Last

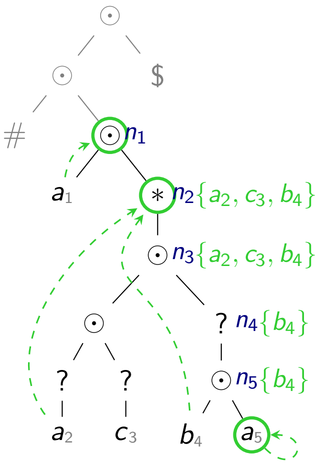


> We compute *SupFirst* pointers in simple traversal.

$b_4 \in First(n_3)$ because
 n_2 ancestor of n_3
 and n_3 ancestor of b_4.

$a_5 \notin First(n_3)$

Work on parse tree: First and Last



We compute *SupFirst* pointers in simple traversal.

$b_4 \in \mathit{First}(n_3)$ because
n_2 ancestor of n_3
and n_3 ancestor of b_4.

$a_5 \notin \mathit{First}(n_3)$

✓ We can test if $a_i \in \mathit{First}(n)$ in constant time.

Symmetrically, we can test if $a_i \in \mathit{Last}(n)$ in constant time.

Reminder

Theorem

We can test if b_j follows a_i in constant time.

Outline

Simple case: k-occurrence expression

Theorem

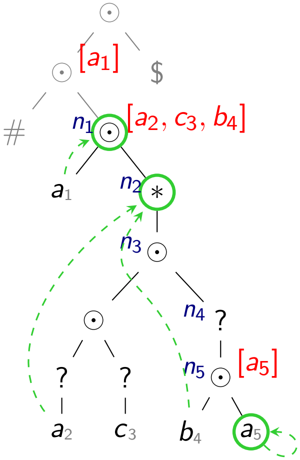
For deterministic k-occurrence expression, the membership problem can be solved in $O(k|w|)$ after $O(|e|)$ preprocessing.

each transition simulated in $O(k)$:

$$e = (ba)^* \boxed{c}\, acabcc?b$$

General case

We put color a in $parent(SupFirst(a_i))$ and store a_i as the witness for color a in that node.



OBSERVATION: positions followed by a_5 are below n_5, those followed by a_2 or b_4 are below n_1...

: a *SupFirst* pointer

$[a_5]$: map $a \mapsto a_5$ (color a, witness a_5)

Finding the right ancestor

We can use "nearest colored ancestor queries".

Nearest colored ancestor [Muthukrishnan et al. 96]

We can preprocess a tree t in expected linear time $O(|t|)$ to answer nearest colored ancestor queries in $O(\log \log |t|)$.

Expected time because of hashmaps, but becomes worst-case linear using lazy arrays.

Evaluation algorithm

Repeatedly jump to the nearest ancestor with color a, and test if its witness follows.

Why is it linear?

Use amortization argument

Summary

	Glushkov	Our results
testing determinism	$O(\|\Sigma\| \times \|e\|)$	$O(\|e\|)$
membership	$O(\|\Sigma\| \times \|e\| + \|w\|)$	$O(\|e\| + \|w\| \log\log(e))$
$\star$ k-occurrence (k-ORE)		$O(k \times \|w\|)$
$\star$ restrictions on $+$		$O(\|e\| + \|w\|)$
$\star$ star free		$O(\|e\| + \|w\|)$

This talk

Future work

- Close the $\log\log$ gap for membership.
- Searching regular pattern instead of matching (KMP...)

Thanks for your attention!

For a few dollars more...

Questions are most welcome!

...but there is no guarantee for the answer