XPath evaluation in linear time with polynomial combined complexity

Paweł Parys

(Common work with Mikołaj Bojańczyk)

XPath is a query language:
XPath queries select nodes in a XML document tree.
We consider fragments called FOXPath and AggXPath

Input: XPath query Q, XML document D
Output: document tree nodes,
which satisfy the query

Contribution: We solve the above problem
1) in time $O(|D|\cdot|Q|^3)$ for Q from FOXPath
2) in time $O(|D|\cdot c^{|Q|})$ for Q from AggXPath

easy corollary

Example document:

```
<html>
  <title>Nice document</title>
  <h1>Section</h1>
  <a href=”http://www.uw.edu.pl/”>University</a>
  <a href=”http://www.google.com/”>Search</a>
  <table><tr><td>
    <a href=”http://www.uw.edu.pl/”>
      A link in a table</a>
  </td></tr></table>
</html>
```

Example document:

```
<html>
  <title>Nice document</title>
  <h1>Section</h1>
  <a href="http://www.uw.edu.pl/">University</a>
  <a href="http://www.google.com/">Search</a>
  <table><tr><td>
    <a href="http://www.uw.edu.pl/">
      A link in a table</a>
  </td></tr></table>
</html>
```

Example query – navigation only (CoreXPath):

```
self::"a" and not (ancestor::"table")
```

Example document:

```
<html>
  <title>Nice document</title>
  <h1>Section</h1>
  <a href="http://www.uw.edu.pl/">University</a>
  <a href="http://www.google.com/">Search</a>
  <table><tr><td>
    <a href="http://www.uw.edu.pl/">
      A link in a table</a>
  </td></tr></table>
</html>
```

Example query – comparing data (FOXPath):

```
self::"a" and ((self/@href=following::"a"/@href)
            or (self/@href=preceding::"a"/@href))
```

Example document:

```
<html>
  <title>Nice document</title>
  <h1>Section</h1>
  <a href="http://www.uw.edu.pl/">University</a>
  <a href="http://www.google.com/">Search</a>
  <table><tr><td>
    <a href="http://www.uw.edu.pl/">
      A link in a table</a>
  </td></tr></table>
</html>
```

Example query – counting (AggXPath):

```
count(preceding)+1=count(root/descendant::"a")
```

Example document:

```
<html>
  <title>Nice document</title>
  <h1>Section</h1>
  <a href="http://www.uw.edu.pl/">University</a>
  <a href="http://www.google.com/">Search</a>
  <table><tr><td>
    <a href="http://www.uw.edu.pl/">
      A link in a table</a>
  </td></tr></table>
</html>
```

Example query – positional arithmetic (full XPath 1.0):

```
descendant[position()=4 and self::"a"]
```

Results summary

CoreXPath (no data)

$O(|D| \cdot |Q|)$ - Gottlob, Koch, Pichler 2002

$O(|D|^{|Q|})$ - real world XPath engines

FOXPath (comparing data)

$O(|D|^2 \cdot |Q|)$ - previous works (GKP)

$O(|D| \cdot c^{|Q|})$, $O(|D| \cdot \log|D| \cdot |Q|^3)$ - Bojańczyk, P. 2008

$O(|D| \cdot |Q|^3)$ - P. 2009

AggXPath (counting)

$O(|D|^2 \cdot |Q|)$ - previous works (GKP)

$O(|D| \cdot c^{|Q|})$ - P. 2009

Full XPath (node positions)

$O(|D|^4 \cdot |Q|^2)$ - Gottlob, Koch, Pichler 2003

Definition of XPath

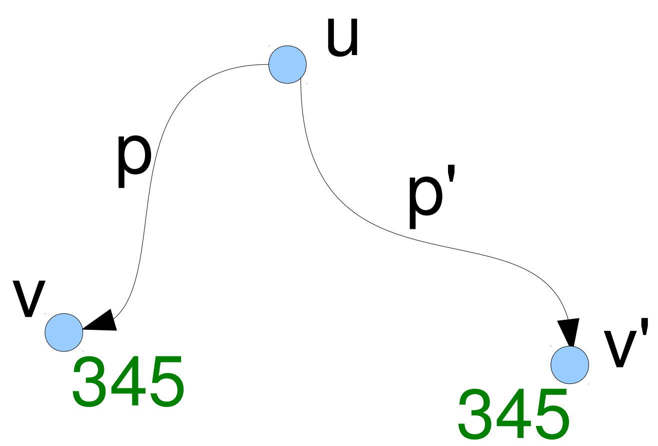
Two types of expressions:

- path expression - returns a set of node pairs:
  - axes: `child`, `parent`, `next-sibling`, `previous-sibling`, `descendant`, `ancestor`, `following-sibling`, `preceding-sibling`
  - [q] - selects a pair (u,u) if u is selected by the node test q
  - composition, union

- node test - returns a set of nodes:
  - name test
  - p - selects a node u if (u,v) is selected by p for some node v
  - $p=p'$ selects a node u if there are (u,v) and (u,v'), selected by p and p' respectively, such that v and v' have the same data value
  - similarly $p \neq p'$, $p<p'$, $p=constant$, etc.
  - or, and, not

most important,
most difficult

Definition of XPath



- $p{=}p'$ selects a node u if there are (u,v) and (u,v'), selected by p and p' respectively, such that v and v' have the same data value
- similarly $p{\neq}p'$, $p{<}p'$, $p{=}constant$, etc.

most important,
most difficult

Algorithm structure

For each node test expression we calculate its value (set of nodes). We do it by induction on the size of the expression:

- name test
- or, and, not

$\Big\}$ easy

- $p=p'$ etc. (selects node u if for some v,v' with the same data value, pair (u,v) is selected by p and pair (u,v') is selected by p'):
  - evaluate all subexpressions $q_1 \ldots q_n$ (node tests)
  - store the results: in the name of every node remember which q_i are satisfied in that node
  - we may assume, that the only atomic path expressions in p and p' are axes and name tests (+ composition, union)

Algorithm idea

Goal: find all nodes satisfying $p=p'$ when the only atomic path expressions in p and p' are axes and name tests.

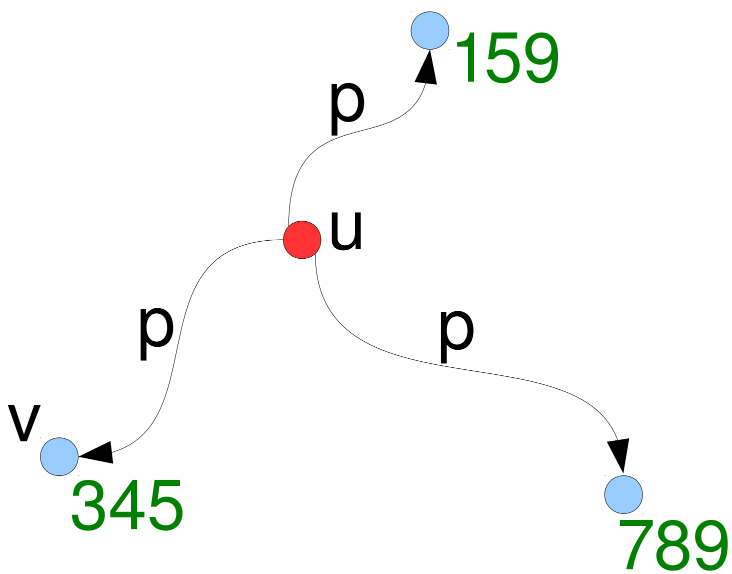
A path expression p may be compiled to a nondeterministic automaton A, which reads a description of a path:

a word over alphabet (node names)$\cup$(one-step axes)

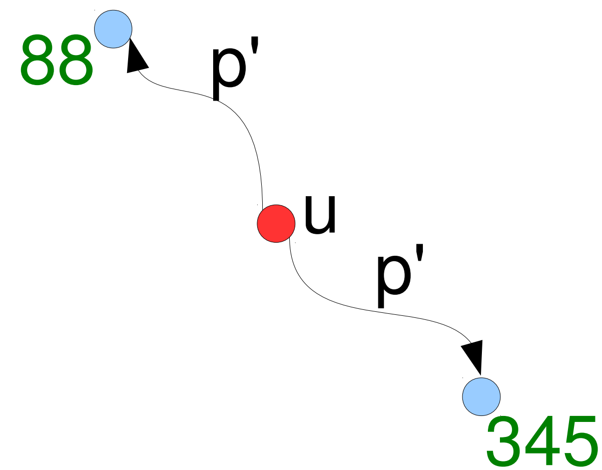
p selects a pair (u,v) iff a description of some path between u and v (not necesarly the shortest path) is accepted by A

Naive approach - quadratic algorithm

For each node u (independently)
calculate all possible values reachable by p:



and by p':

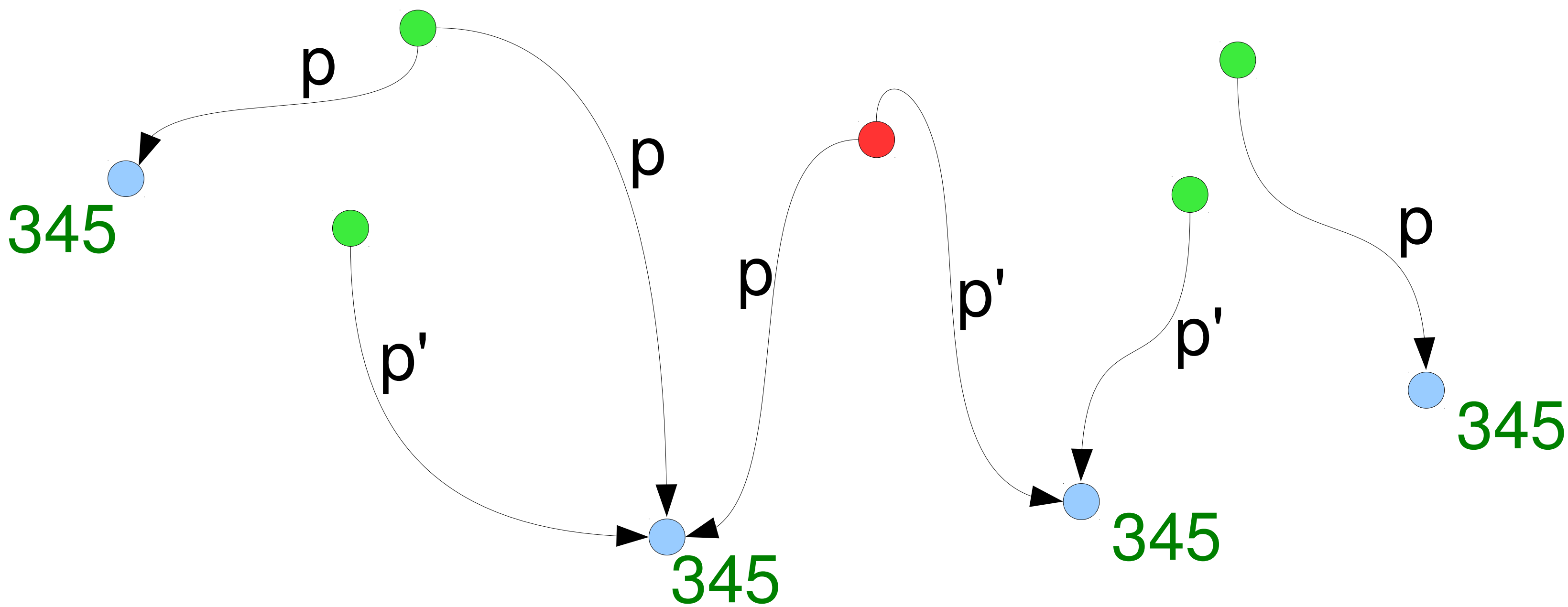


and check if these two sets have nonempty intersection.

For each u it can be done in linear time.
Thus the whole algorithm is quadratic.

Second naive approach - quadratic algorithm

For each data value (independently)
mark the nodes u which can reach this data value by p and p'



For each data class it can be done in linear time.
Thus the whole algorithm is quadratic.

We will improve this approach!

Algorithm idea

Goal: find all nodes satisfying $p=p'$ when the only atomic path expressions in p and p' are axes and name tests.

A path expression p may be compiled to a nondeterministic automaton A, which reads a description of a path:

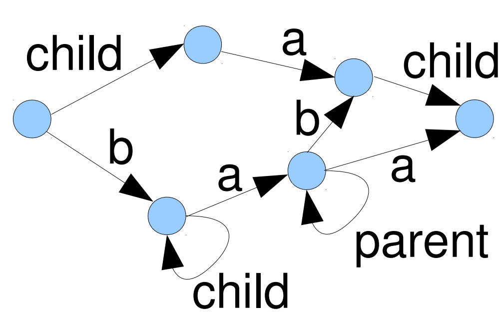
a word over alphabet (node names)$\cup$(one-step axes)

p selects a pair (u,v) iff a description of some path between u and v (not necesarly the shortest path) is accepted by A

But p is not an arbitrary regular expression,

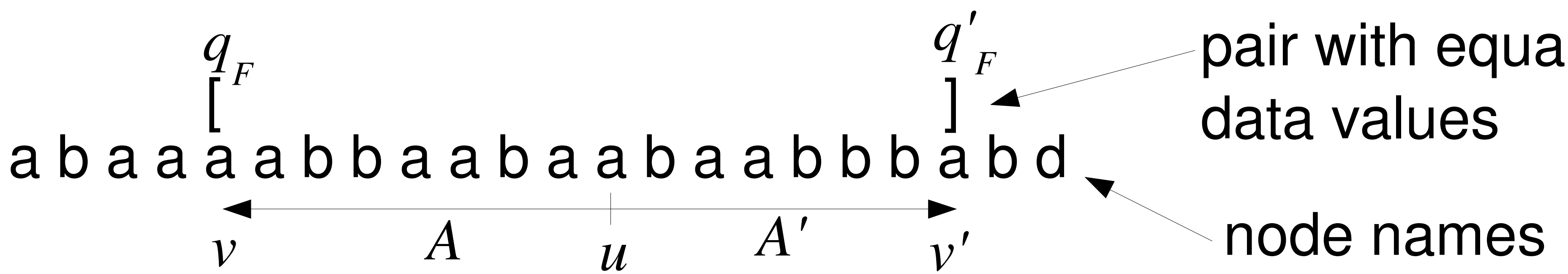
there is no Kleene star in XPath!!!!

So the automaton has only trival cycles (reading axes):



Algorithm idea - special case

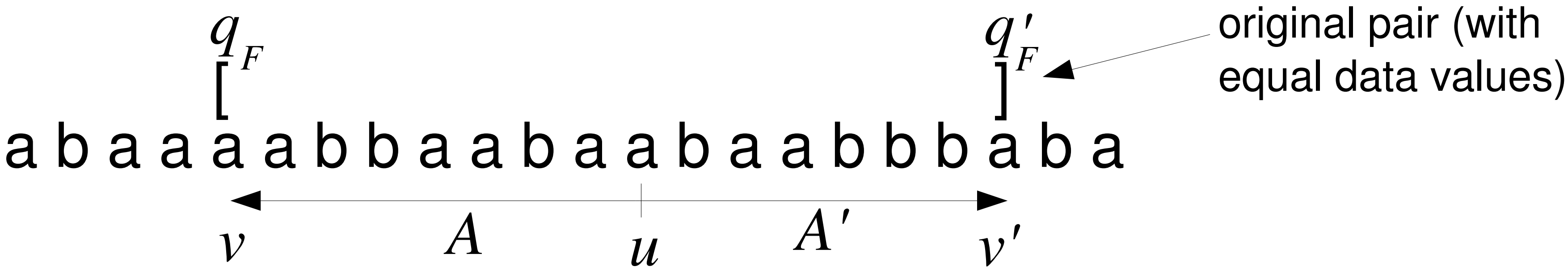
- assume we have only a word with data (instead of a tree)
- automaton A for p goes only to the left and A' for p' only to the right
- every data value appears in exactly two places
  (denoted by a pair of brackets)



We have to mark all such u.

We will replace this set of bracket pairs by another one from which it is easier to calculate the selected u.

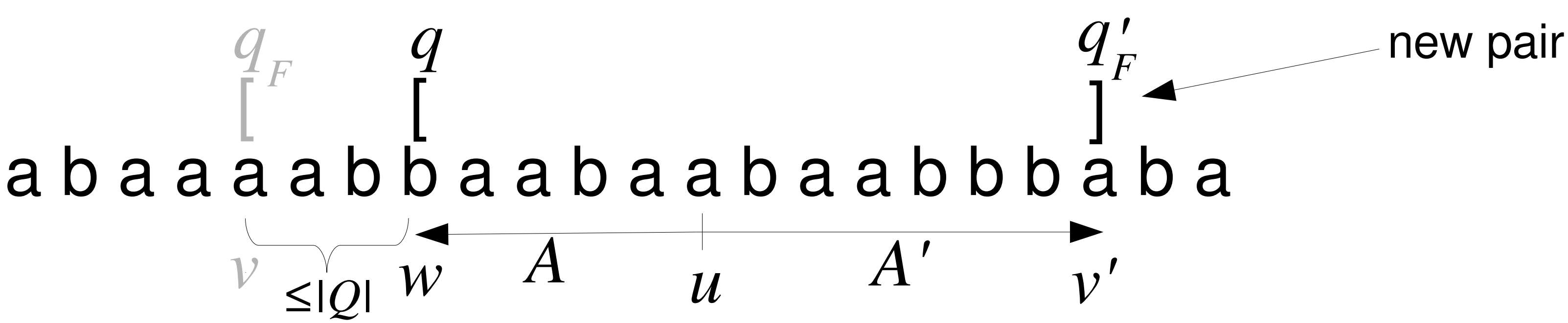
Algorithm idea - special case, continued



The automaton A in some of last $|Q|$ positions has to visit a state q with a loop reading `left`.

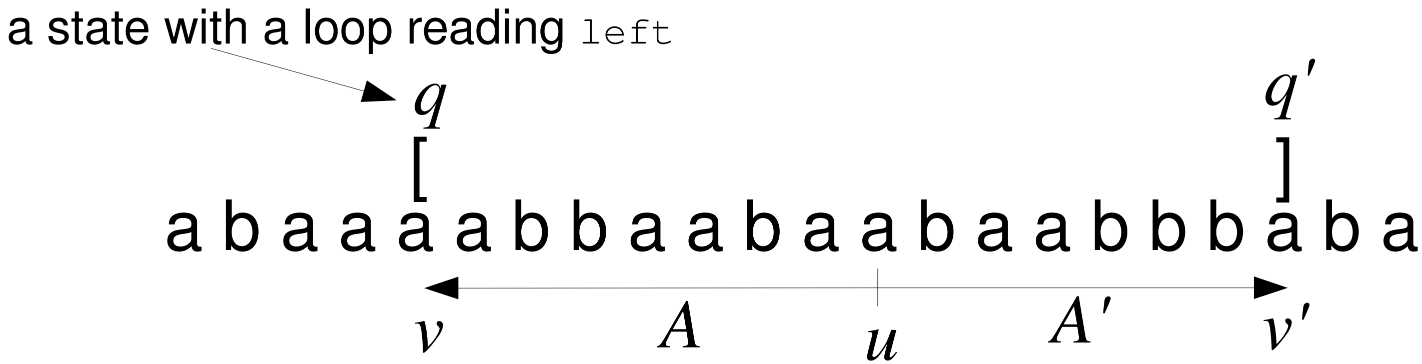
We may replace this pair of brackets by at most $|Q|^2$ new pairs:

- from state q in w we may reach q_F in v,
- distance between w and v is at most $|Q|$
- state q has a loop reading `left`.



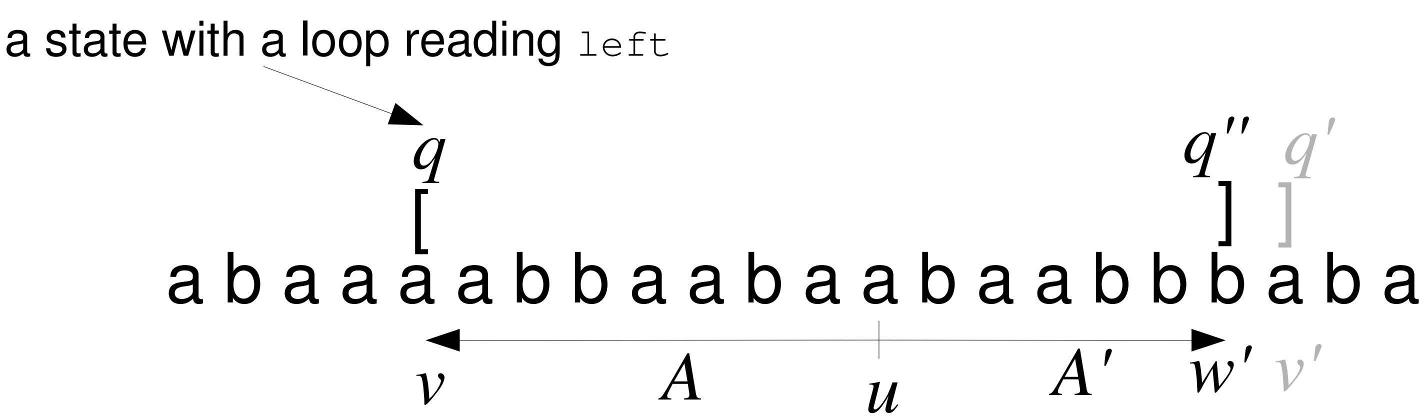
(possibly we should also mark nodes u close to v, if starting from u we may reach q_F in v and q'_F in w)

Algorithm idea - special case, step 2



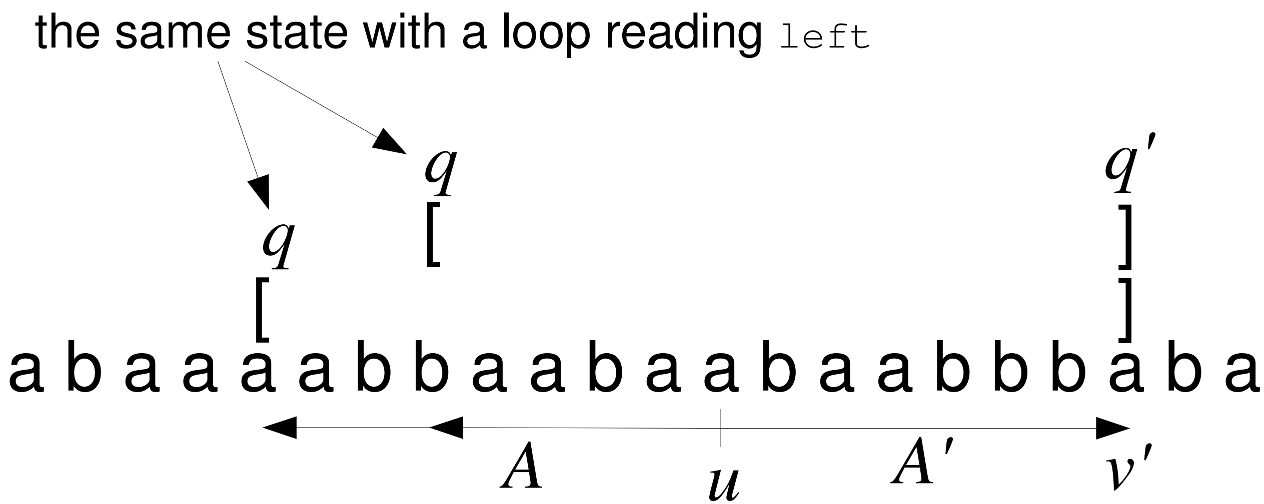
Starting from the end of the word we move brackets to the left:

- we move right bracket at v' one node to the left (changing the state)
- moreover, if $q'=q_0$ and starting at v', A reaches q at position v, then v' should be marked as u



This creates $|Q|$ new pairs, which have to be processed again and again, but...

Observation



The closer pair may be removed,
it generates the same nodes u.

So for every node v' there may be at most $|Q|^2$ pairs of brackets, one for every pair of states.

Final lemma

What is missing to solve the special case:

For given u,v,q_0,q (where q has a loop reading `left`)

check if A may reach q in v starting from q_0 in u.

Equivalent question:

For given u,q_0,q (where q has a loop reading `left`) where is the rightmost v such that A (going left) may reach q in v starting from q_0 in u. We call that $first(u,q_0,q)$.

This information may be calculated in one left-to-right pass:

- It is possible that $first(u,q_0,q)=u$
- Otherwise it is the rightmost of $first(u',q',q)$ for q' which may be reached in u' from q_0 in u (where u' is the node one step to the left)

Algorithm idea - more general case

- assume we have only a word with data (instead of a tree)
- automaton A for p goes only to the left and A' for p' only to the right
- a data value can appear in any number of places (v_1,v_2,v_3)

$$
\begin{array}{cccccccccccccccccccccc}
 & & & & q_F & & & & & & & & Q'_2 \; Q_2 & & & & & & & q'_F & & \\
 & & & & [& & & & & & & &][& & & & & & &] & & \\
a & b & a & a & a & a & b & b & a & a & b & a & a & b & a & a & b & b & b & a & b & d \\
 & & & & v_1 & & & & & & & & v_2 & & & & & & & v_3 & &
\end{array}
$$

Create a bracket pair for consecutive nodes with the same data.

$Q_2 = \{q_F\} \cup \{q$: from q in v_2, A can reach q_F in $v_1\}$

$Q_2' = \ldots$

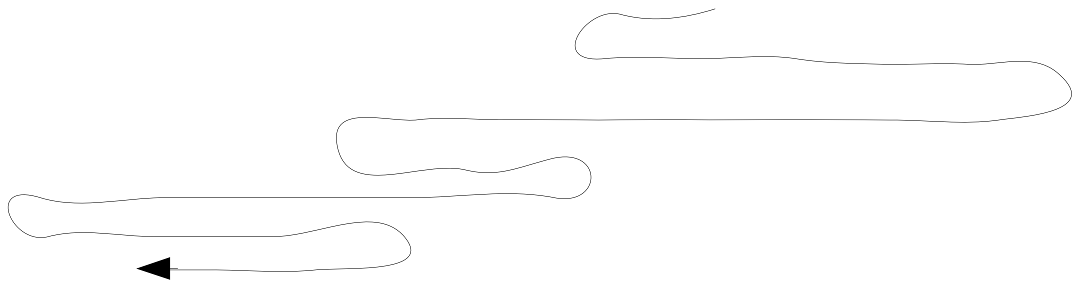
Continue like previously.

The total number of brackets is linear.

After precomputation it can be read in constant time (previous slide)

Algorithm idea - more general case

- assume we have only a word with data (instead of a tree)
- automata A and A' can make loops



Precalculate the loops:

$$\mathrm{loops}(v) = \{(p,q) : \text{from } p \text{ in } v, A \text{ can reach } q \text{ in } v\}$$
$$= (\mathrm{loops}_{\mathrm{left}}(v) \cup \mathrm{loops}_{\mathrm{right}}(v))^*$$

Add the $\mathrm{loops}(v)$ set to the label of v.

Construct a new automaton, which goes only left and instead of making a loop in v, it reads $\mathrm{loops}(v)$.

Algorithm idea - the general case (trees)

Class of a data value = nodes with these data value,
and their least common ancestors