XPath evaluation in linear time with polynomial combined complexity

Paweł Parys

We consider a problem of evaluating XPath query in an XML document:

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

Contribution: The above problem may be solved in time $O(|D|\cdot|Q|^3)$ for Q from a fragment of XPath called FOXPath

Which fragment?

- navigation
- comparing data
  (query $\alpha = \beta$, satisfied in nodes x such that some (x, y_1) is selected by α and some (x, y_2) is selected by β and data value in y_1 and y_2 is the same)
- we do not allow counting and positional arithmetic

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

$O(|D|\cdot|Q|^{3})$ - this result

Full XPath (counting, node positions)

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

Contribution

Why is this algorithm better than the previous one?

- better complexity in query size
- deals with <, <=, >, >=, not only with = and !=
- complexity linear in
  (number of bytes of input + size of alphabet)
  instead of (number of bits of input)
- deals with text nodes, not only attribute values
  (not trivial, XPath says: text value of an element node is a concatanation of all its text descendants - so the total length of text values may be quadratic in input size)
- easier to understand

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

} 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 $q_1 \dots q_n$ (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

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:

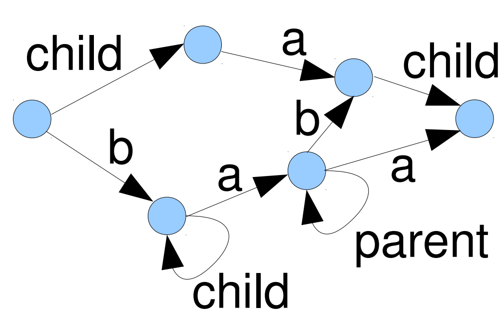
a word over alphabet (node names)$\cup$(one-step axes)

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,

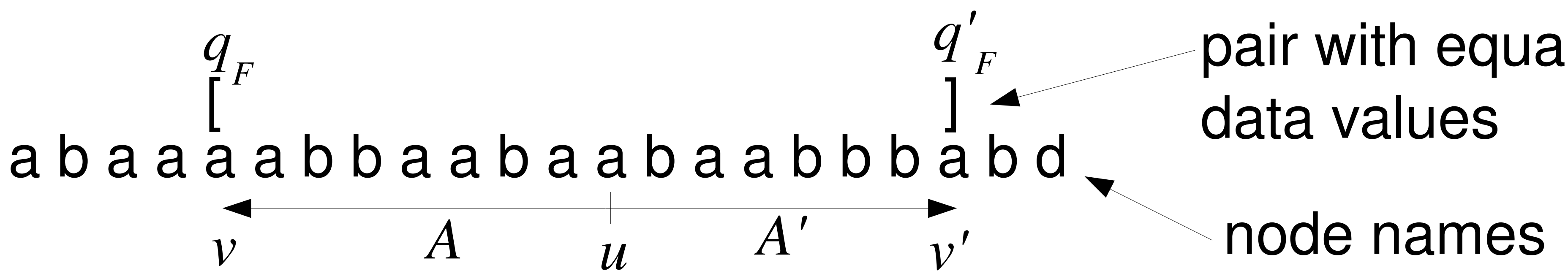
there is no Kleene star in XPath!!!!

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



Algorithm idea - special case

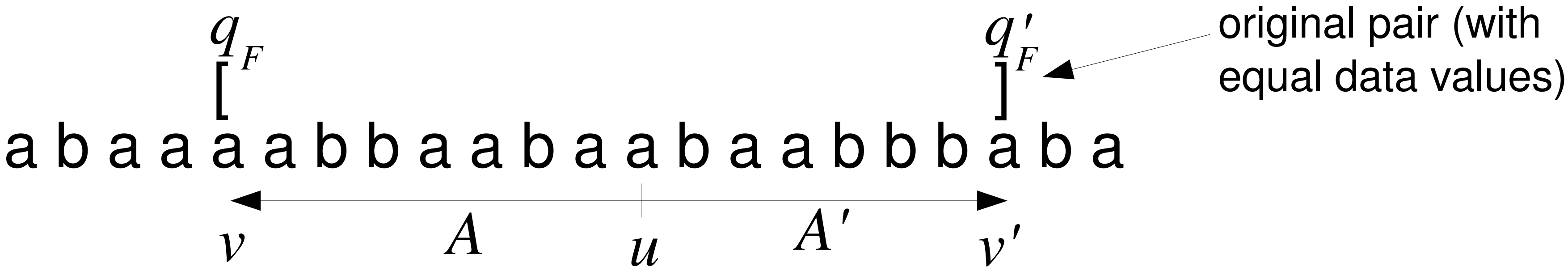
- assume we have only a word with data (instead of a tree)
- automaton A for p goes only to the left and A' for p' only to the right
- every data value appears in exactly two places
  (denoted by a pair of brackets)



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.

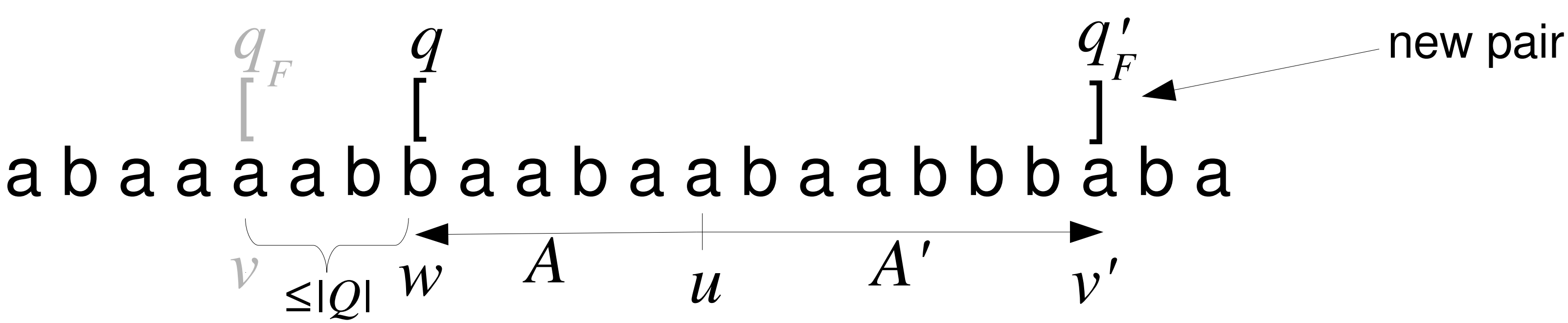
Algorithm idea - special case, continued



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

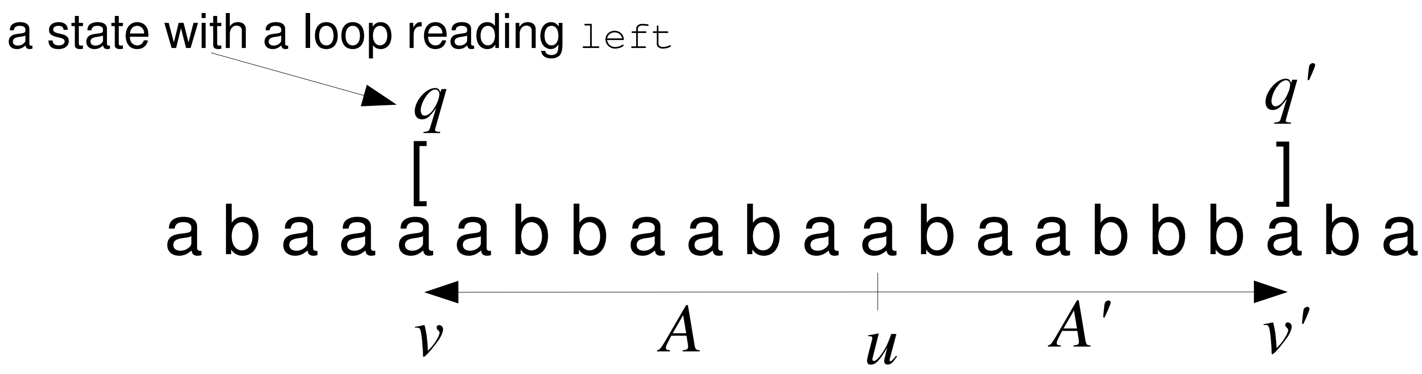
We may replace this pair of brackets by at most $|Q|^2$ new pairs:

- 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`.



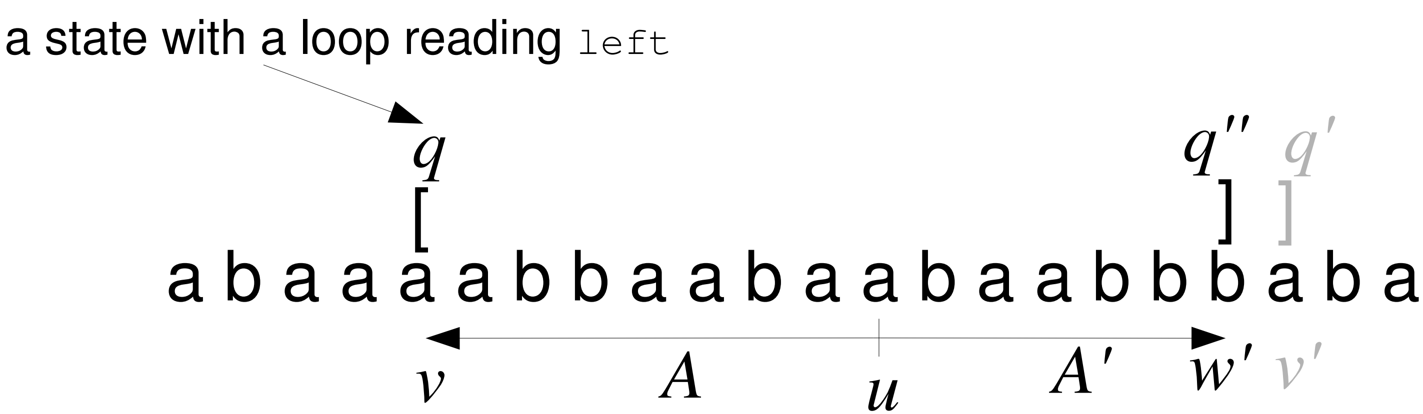
(possibly we should also mark nodes u close to v, if starting from u we may reach q_F in v and q'_F in w)

Algorithm idea - special case, step 2



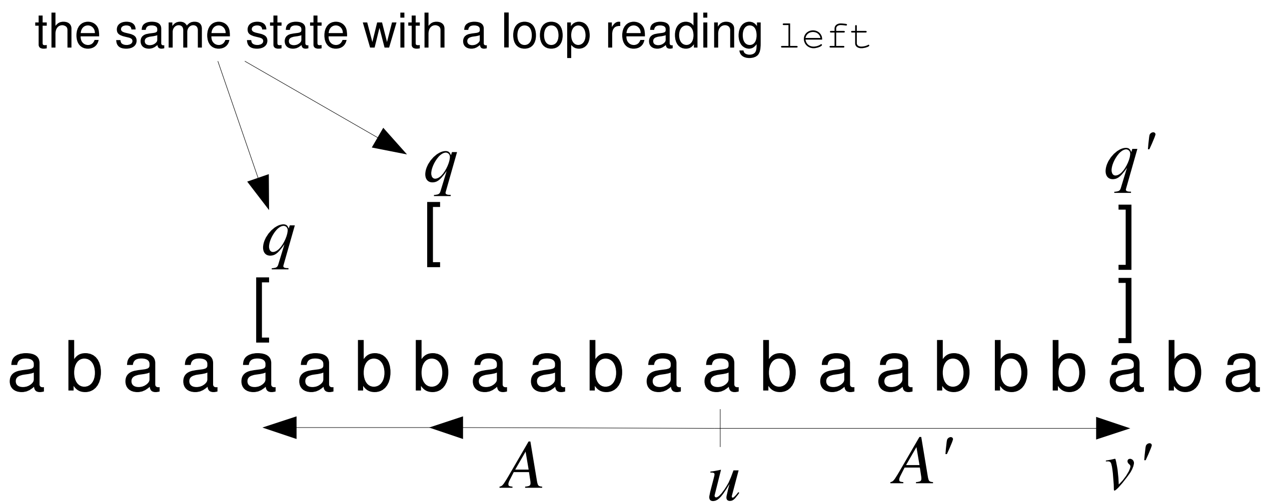
Starting from the end of the word we move brackets to the left:

- we move right bracket at v' one node to the left (chaning the state)
- or $q'=q'_0$ and starting at v', A reaches q at position v (then we mark v')



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 righmost v such that A 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-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)