Tree-walking automata

Mikołaj Bojańczyk
(Warsaw University)

Plan

-A tree-walking automaton

-Expressive power

-Pebble automata and logic

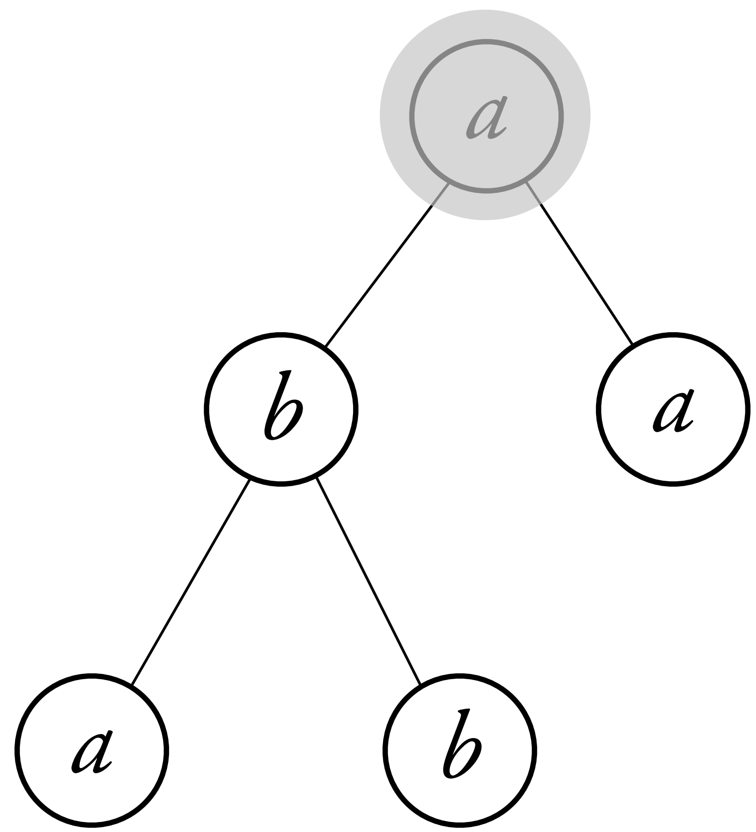
Plan

-A tree-walking automaton

-Expressive power

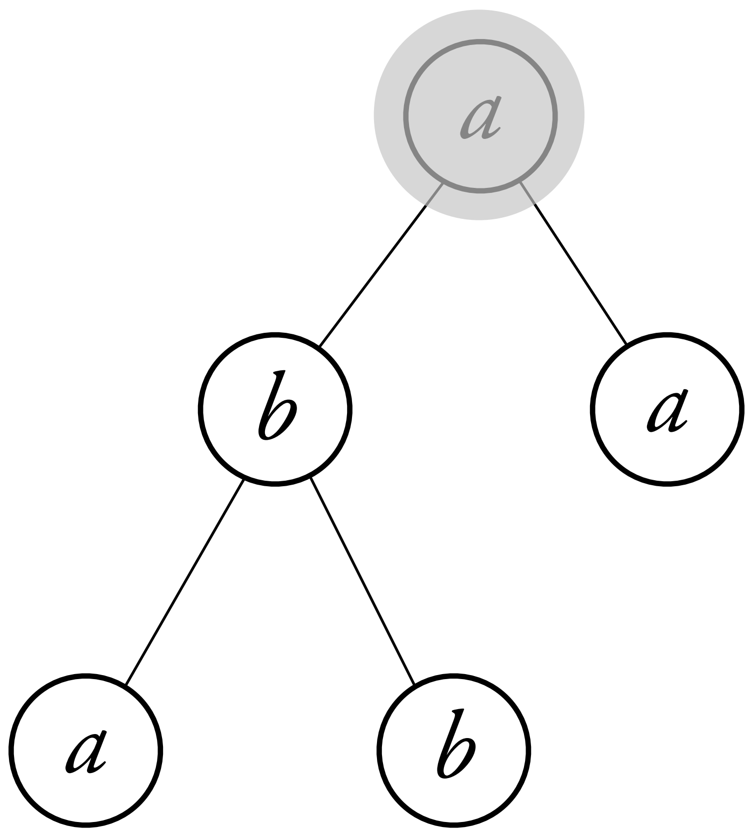
-Pebble automata and logic

Trees are finite, binary and labeled



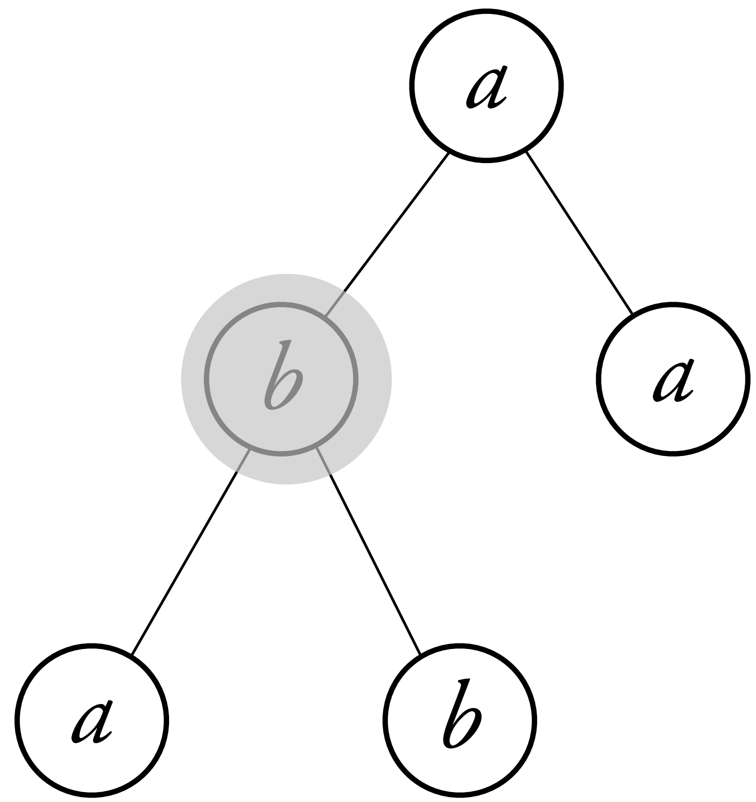
Trees are finite, binary and labeled

A tree-walking automaton is sequential and two-way.



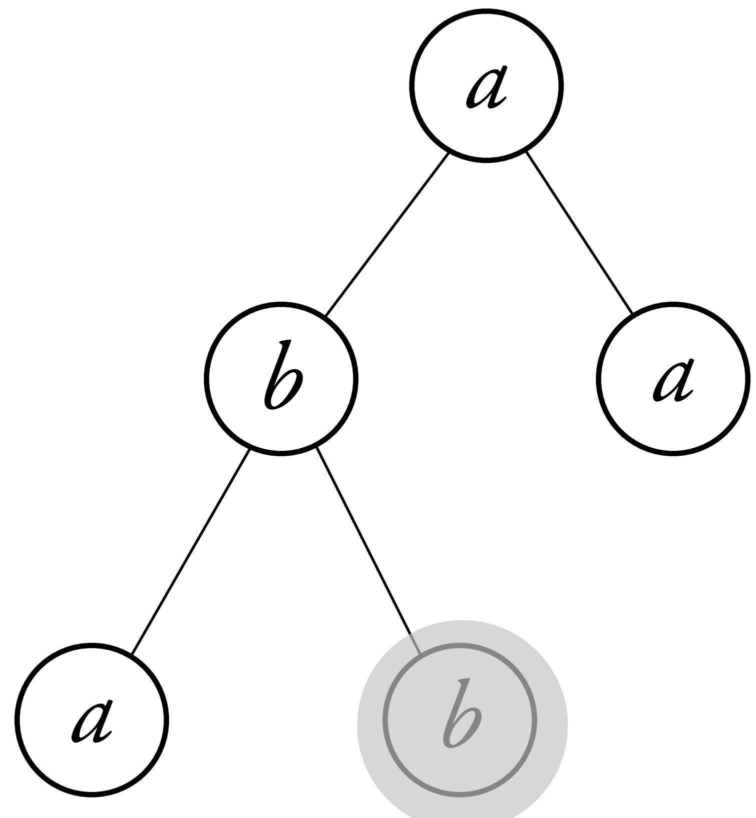
Trees are finite, binary and labeled

A tree-walking automaton is sequential and two-way.



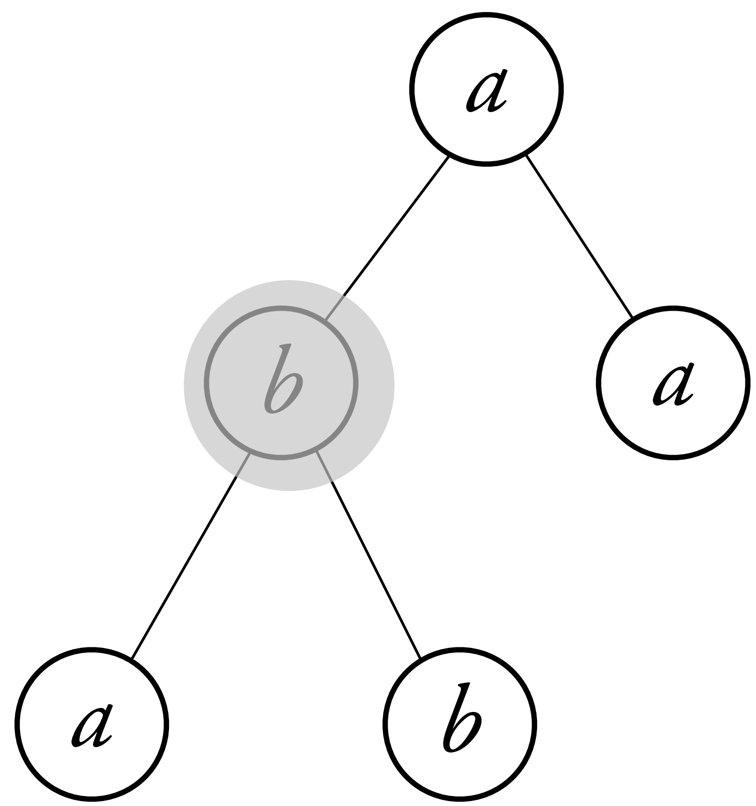
Trees are finite, binary and labeled

A tree-walking automaton is sequential and two-way.



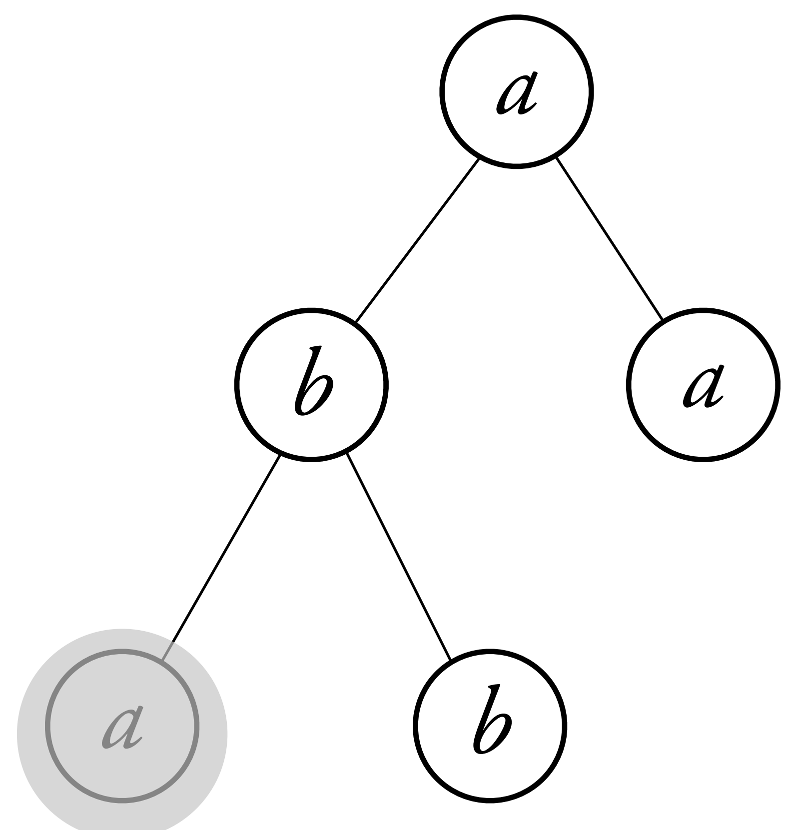
Trees are finite, binary and labeled

A tree-walking automaton is sequential and two-way.

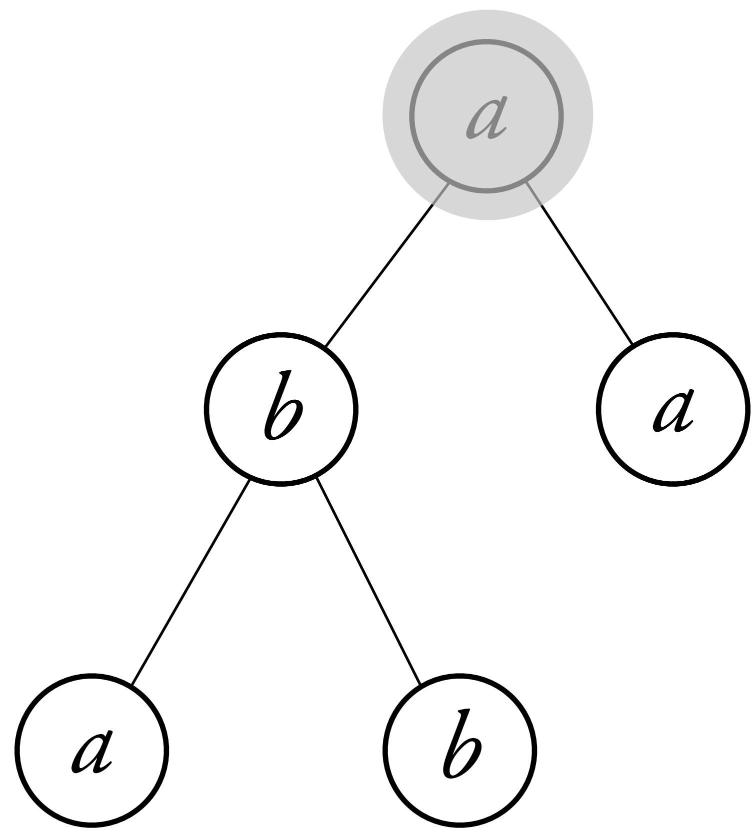


Trees are finite, binary and labeled

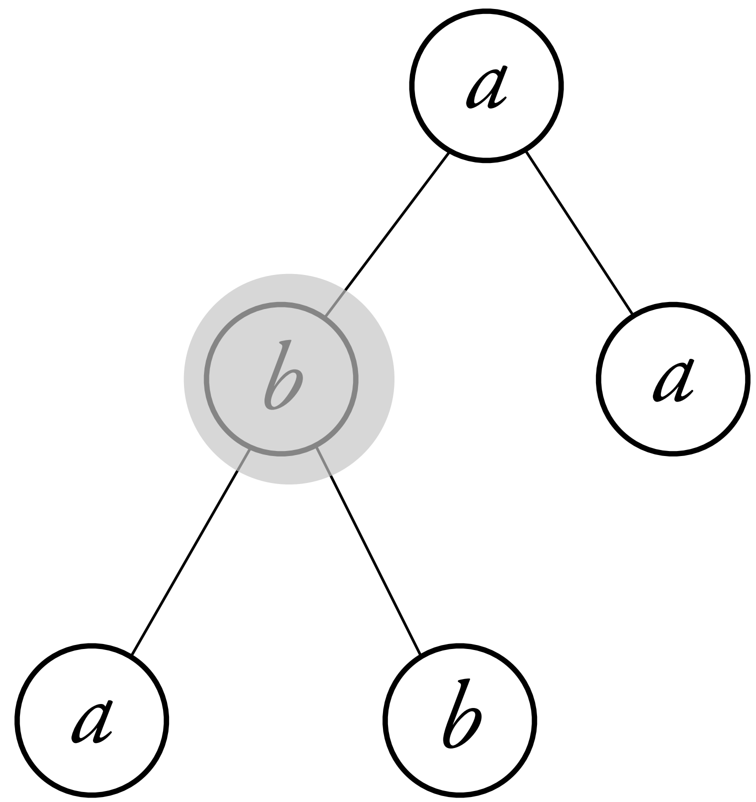
A tree-walking automaton is sequential and two-way.



If the state is p and the node is the root with label a, then move to the left child and change state to q.



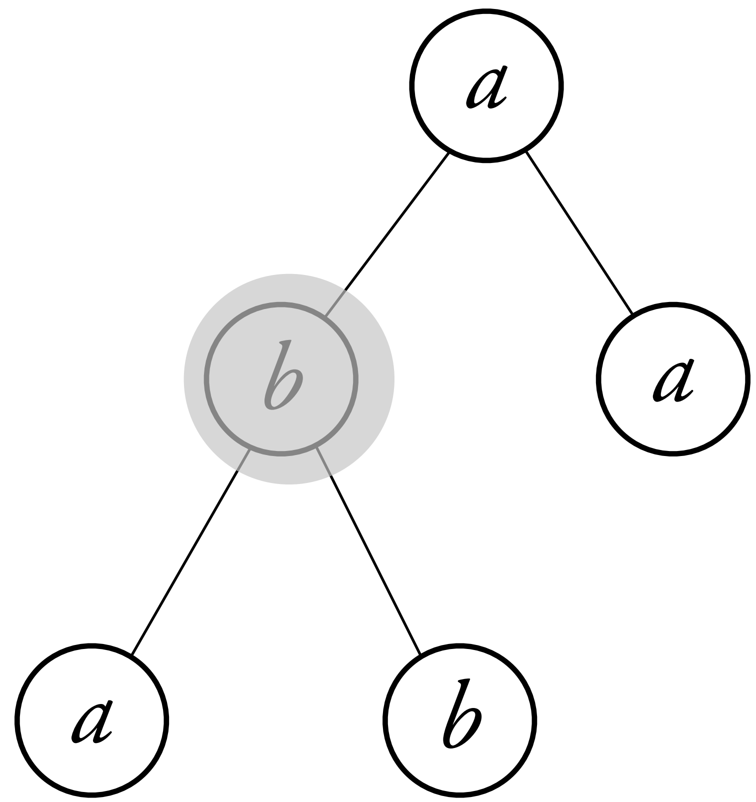
If the state is p and the node is the root with label a, then move to the left child and change state to q.



test

If the state is p and the node is the root with label a, then move to the left child and change state to q.

command

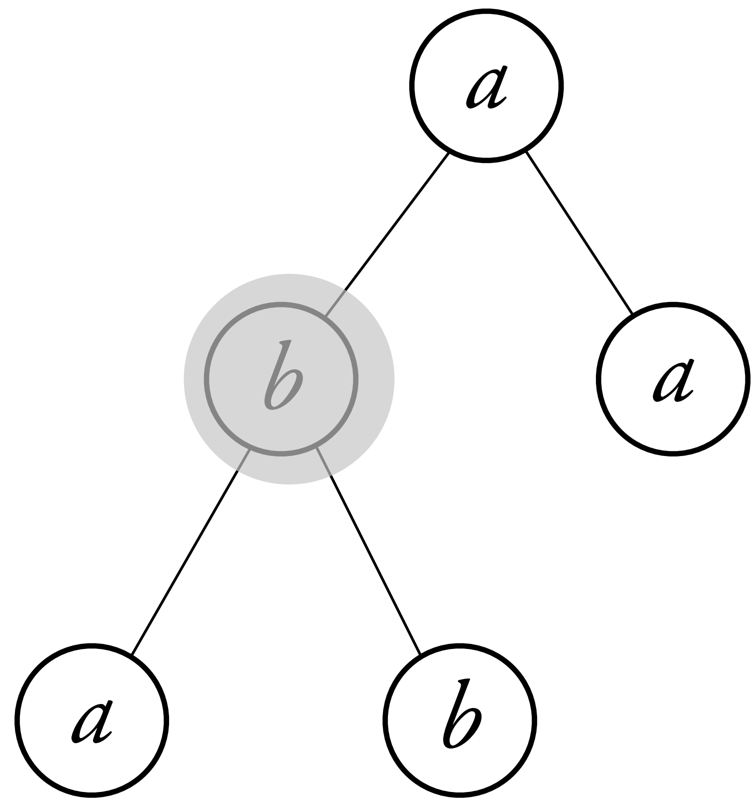


test

If the state is p and the node is the root with label a, then move to the left child and change state to q.

command

Tests are boolean combinations of: has label a, is right/left child, is leaf



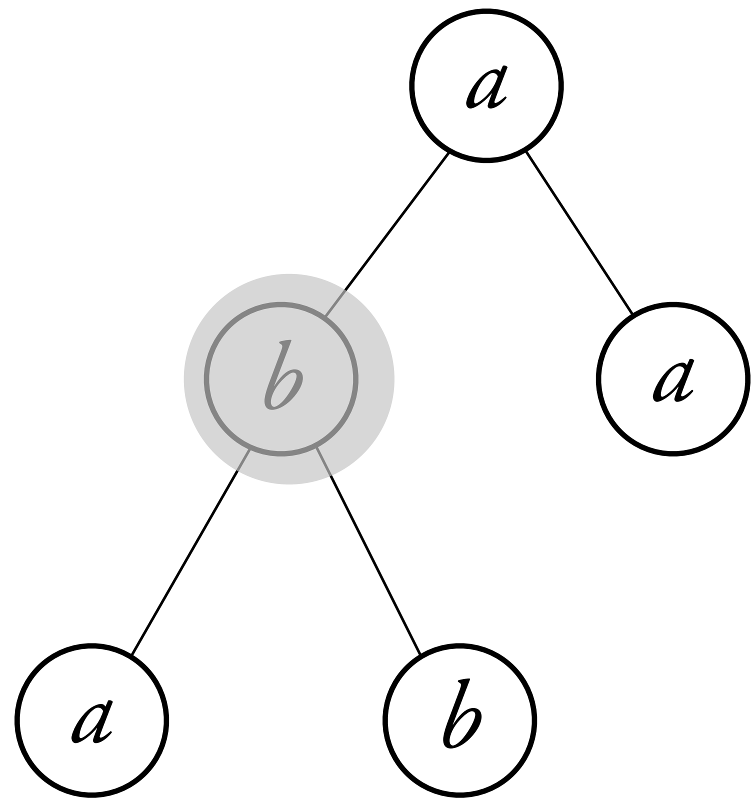
Commands are: move left/right/up, accept, reject

test

If the state is p and the node is the root with label a, then move to the left child and change state to q.

command

Tests are boolean combinations of:
has label a,
is right/left child,
is leaf



Commands are:
move left/right/up,
accept,
reject

Def. A tree walking-automaton is a tuple $\langle Q, q_I, \Sigma, \delta \rangle$

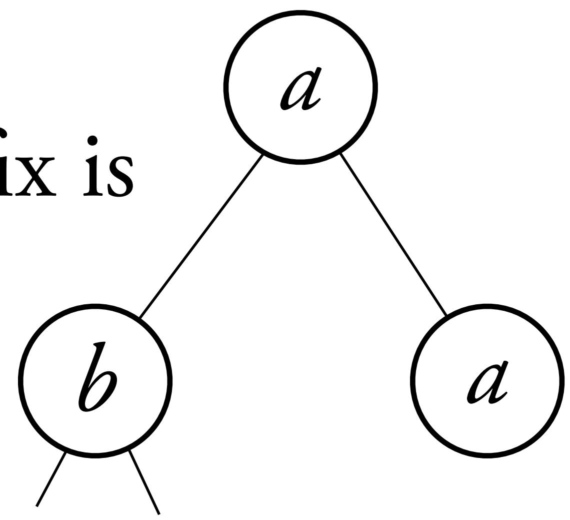
states

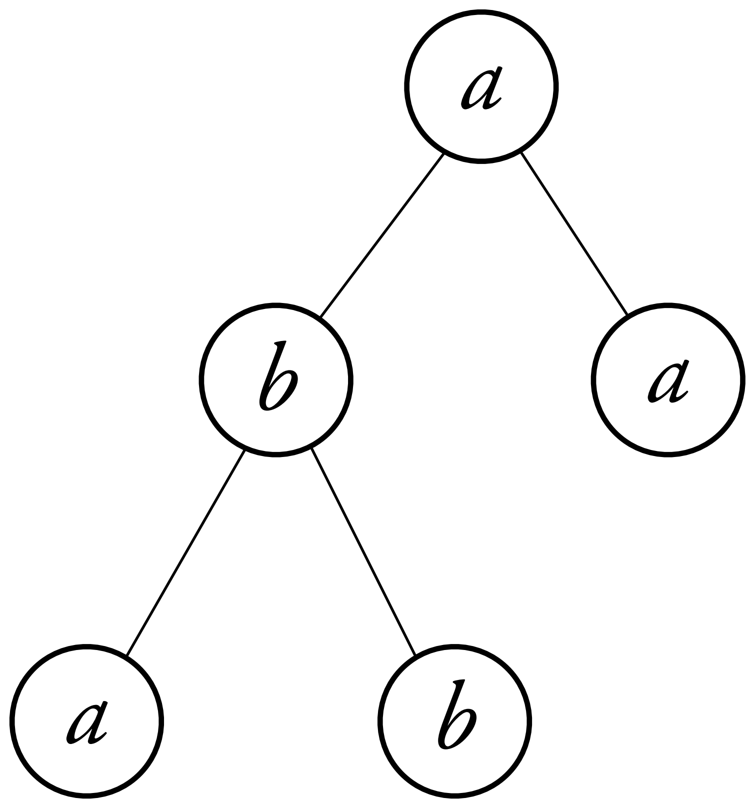
initial state

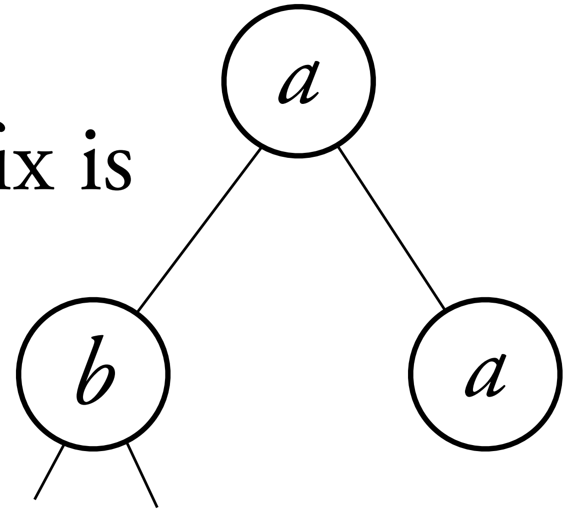
alphabet

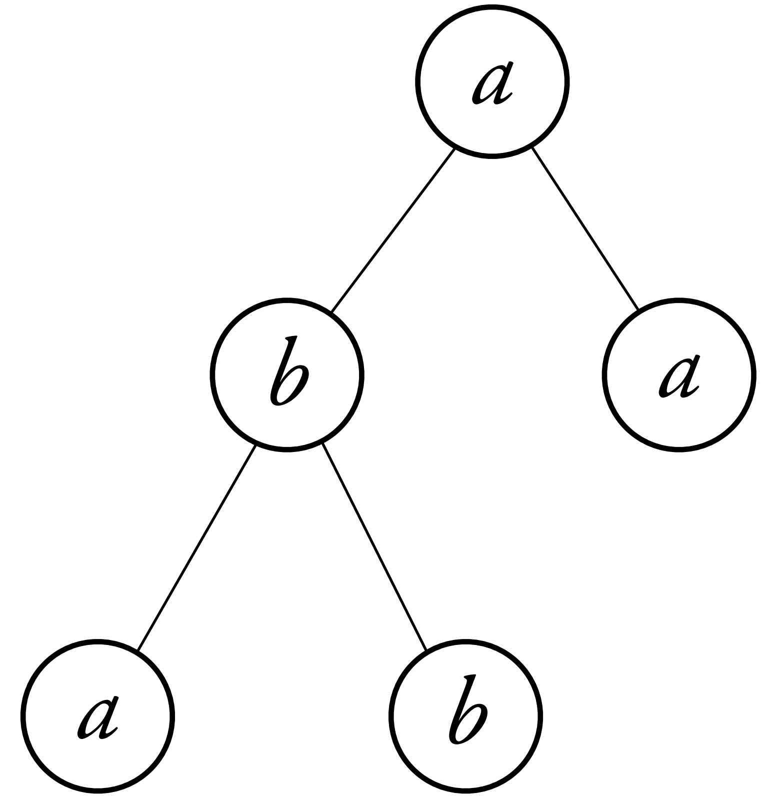
transitions

Example.
Check if the prefix is



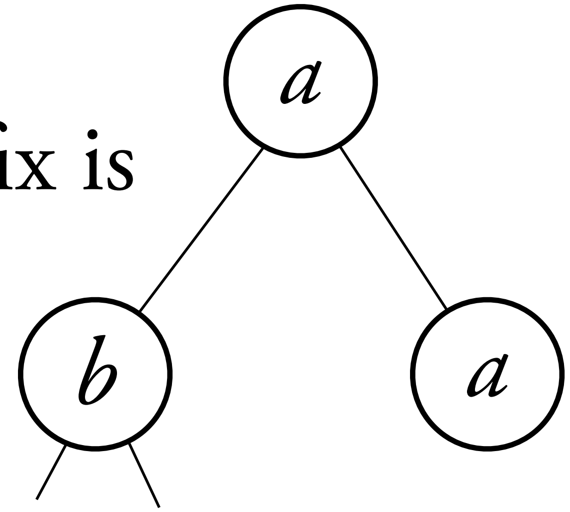


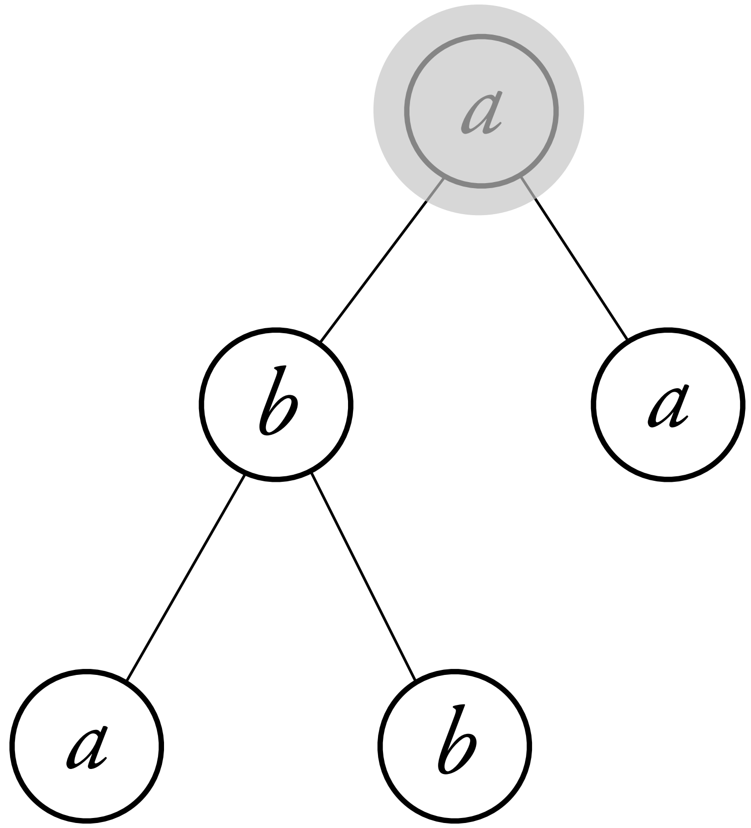
Example.
Check if the prefix is 



In state p, label a and root, move left, change state to q
In state q, label b and not leaf, move up, change state to r
In state r, move right, change state to s
In state s, label a and leaf, accept.

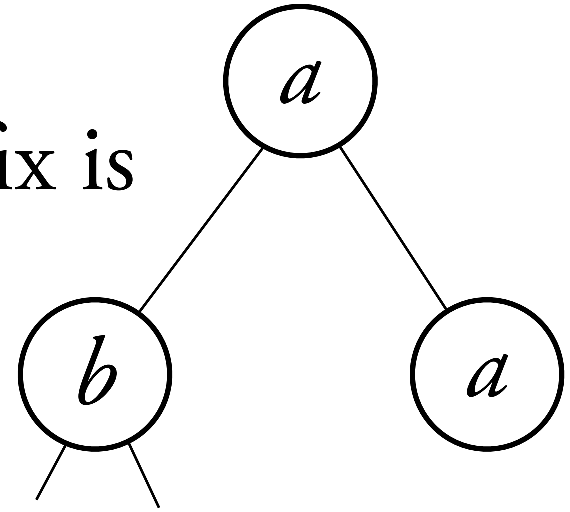
Example.
Check if the prefix is

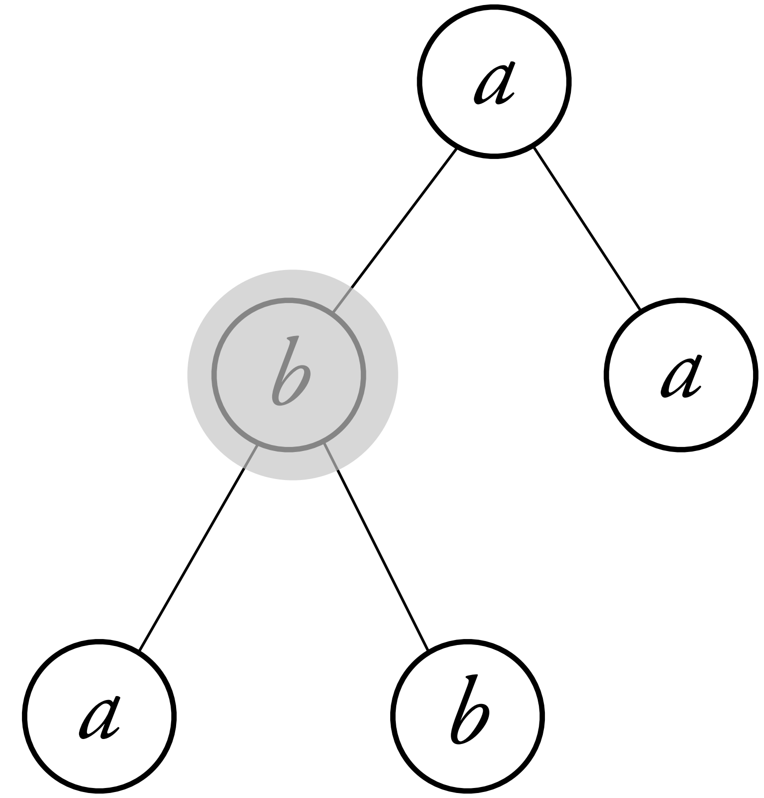




In state p, label a and root, move left, change state to q
In state q, label b and not leaf, move up, change state to r
In state r, move right, change state to s
In state s, label a and leaf, accept.

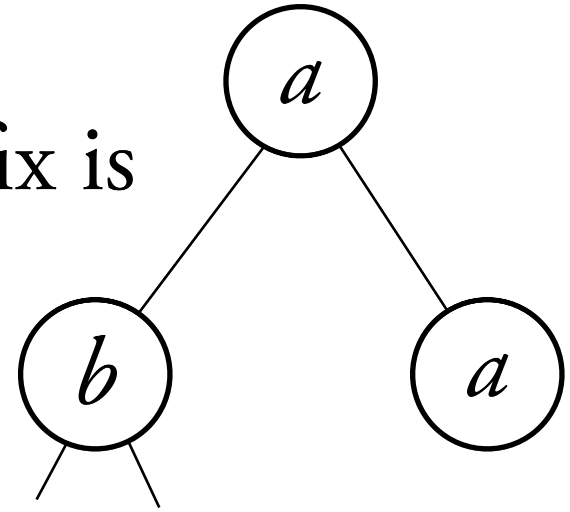
Example.
Check if the prefix is

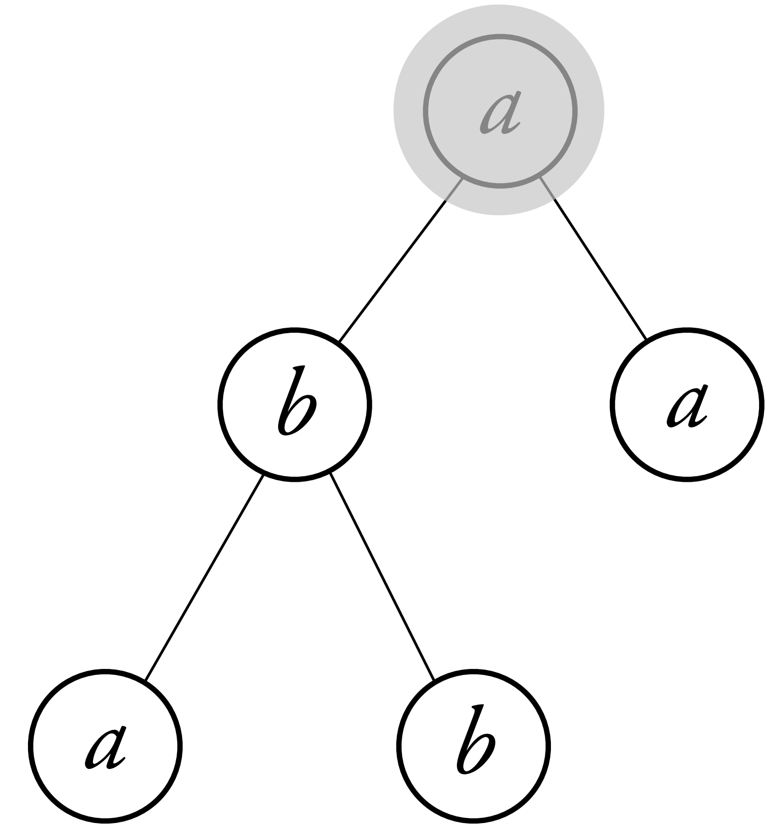




In state p, label a and root, move left, change state to q
In state q, label b and not leaf, move up, change state to r
In state r, move right, change state to s
In state s, label a and leaf, accept.

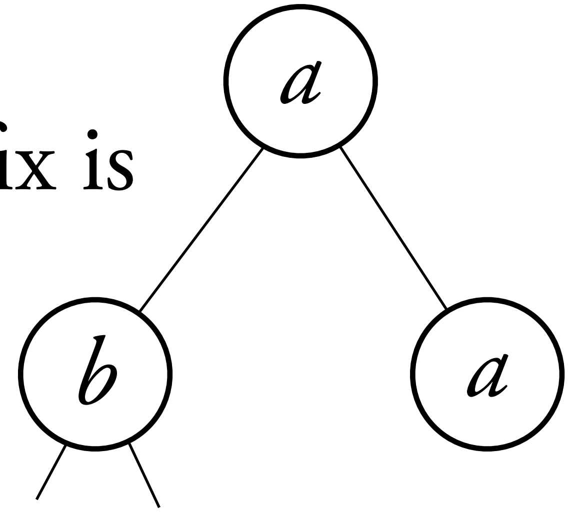
Example.
Check if the prefix is

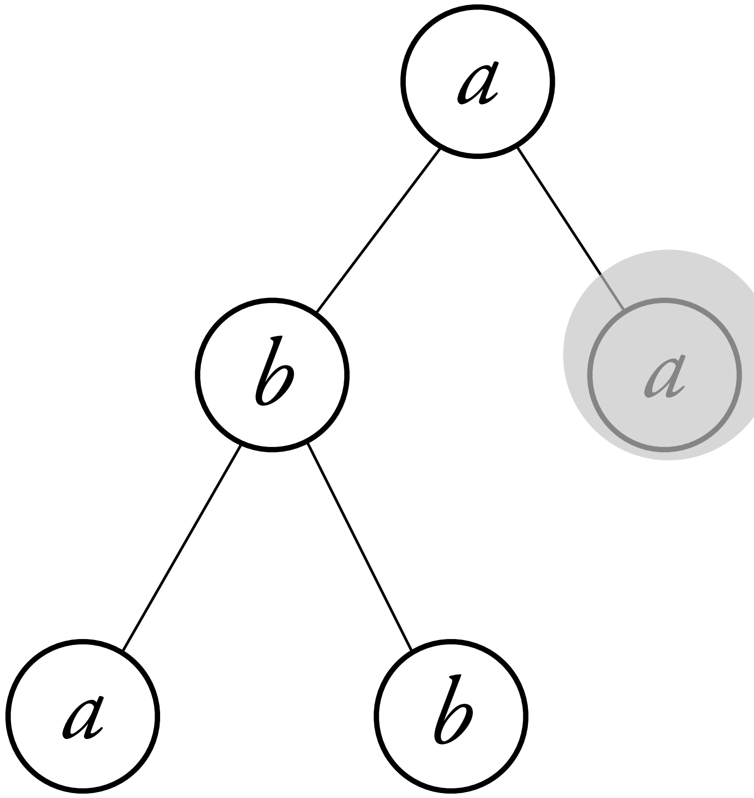




In state p, label a and root, move left, change state to q
In state q, label b and not leaf, move up, change state to r
In state r, move right, change state to s
In state s, label a and leaf, accept.

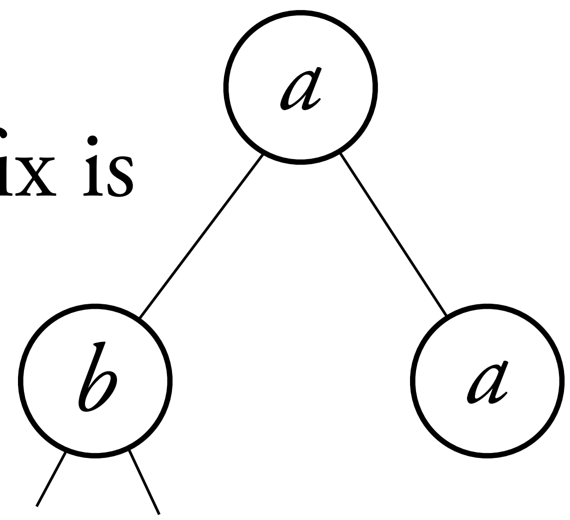
Example.
Check if the prefix is

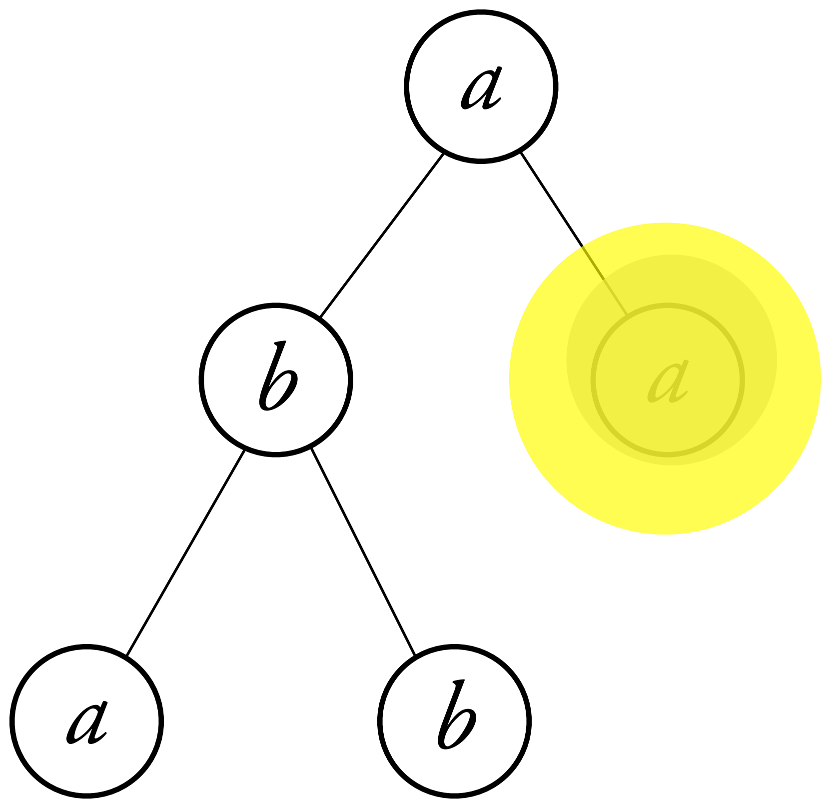




In state p, label a and root, move left, change state to q
In state q, label b and not leaf, move up, change state to r
In state r, move right, change state to s
In state s, label a and leaf, accept.

Example.
Check if the prefix is

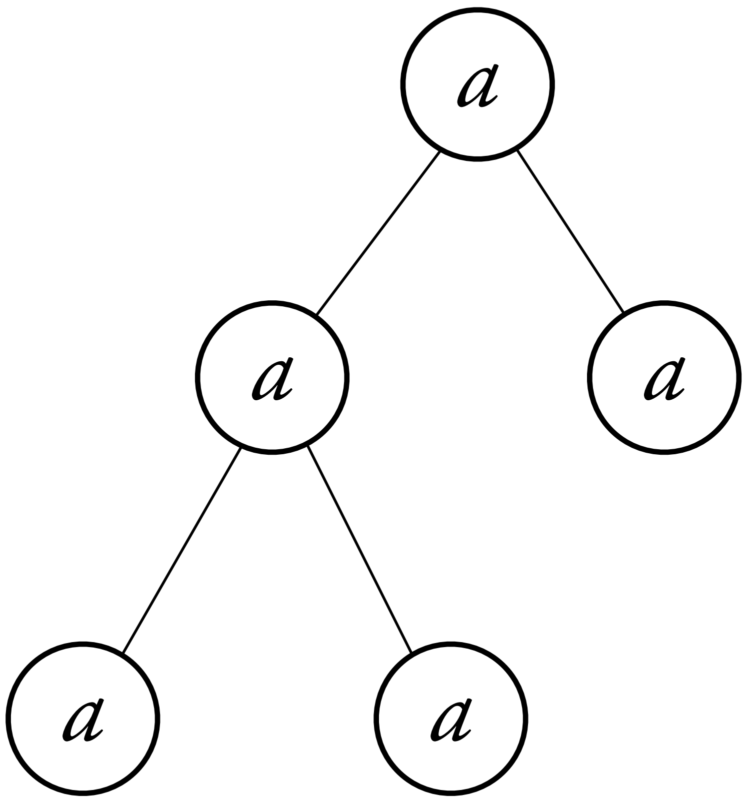




In state p, label a and root, move left, change state to q
In state q, label b and not leaf, move up, change state to r
In state r, move right, change state to s
In state s, label a and leaf, accept.

Example.
Some node has label b

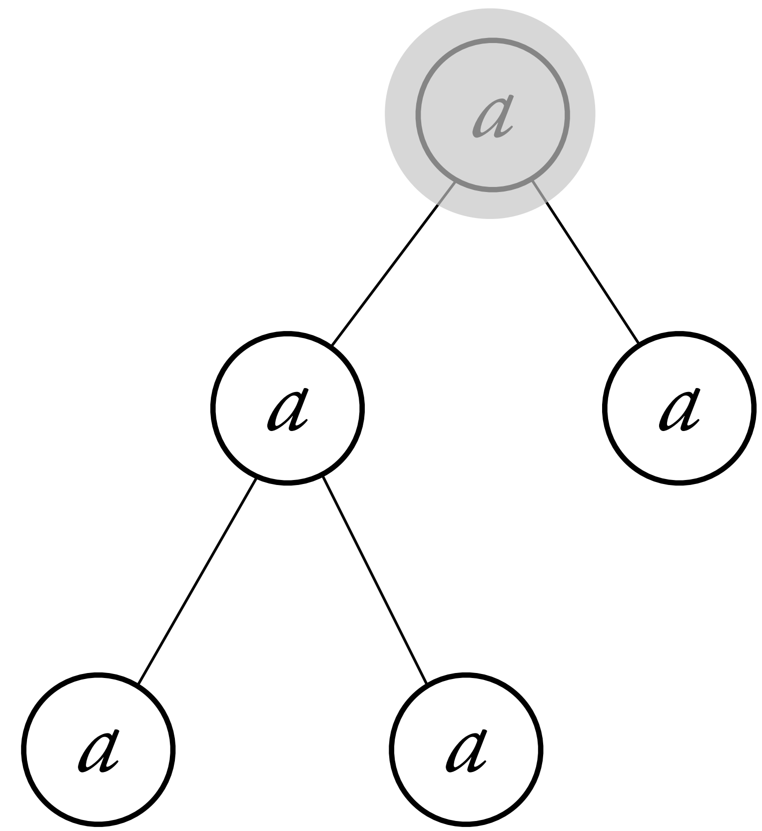
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label *b*

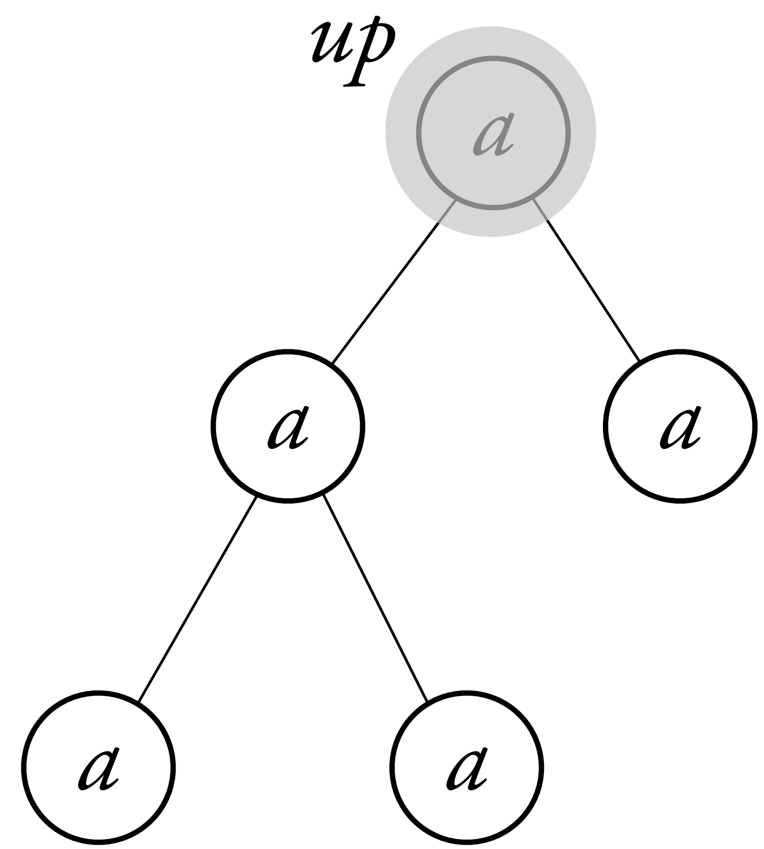
In state *p*, move left.
In state *p*, move right.
In state *p*, label *b*, accept.



Example.
All nodes have label *a*

Example.
Some node has label b

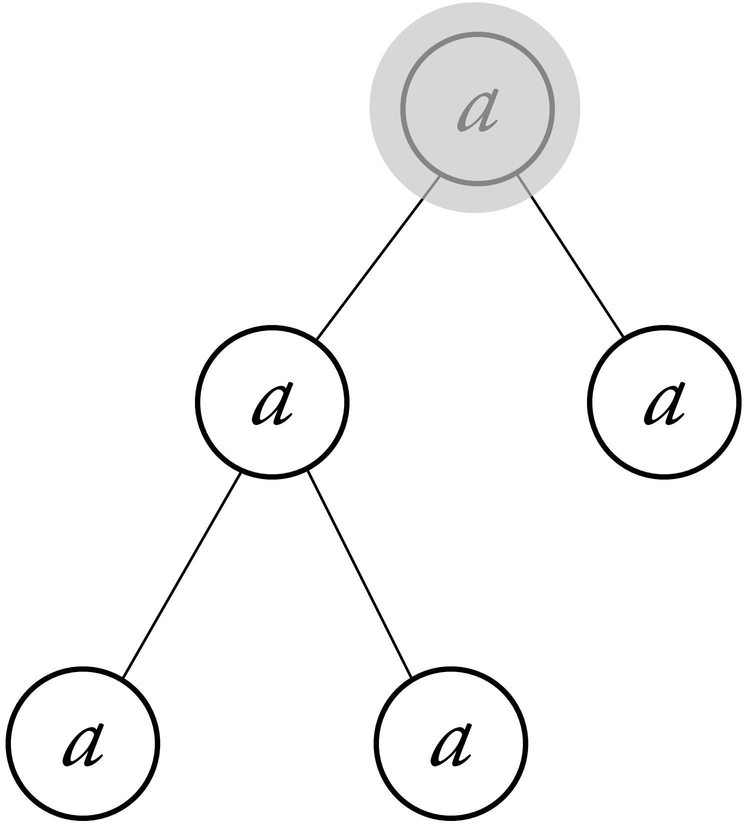
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

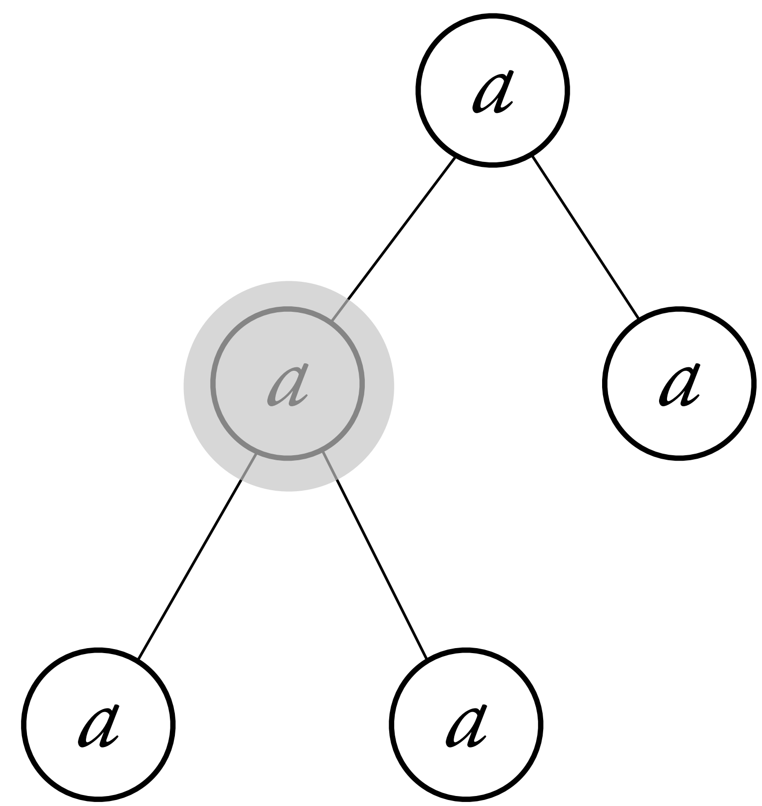
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

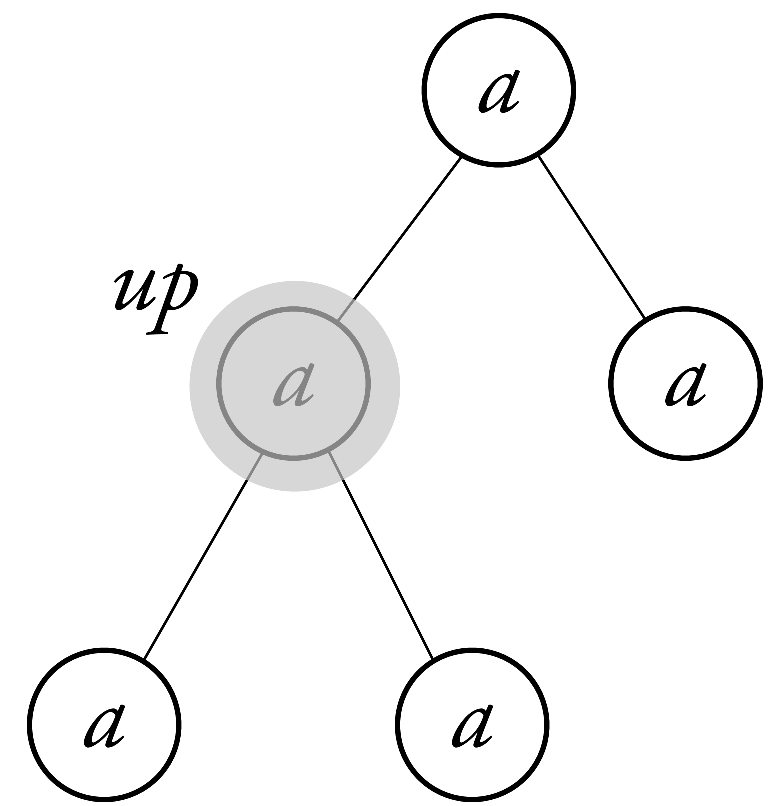
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

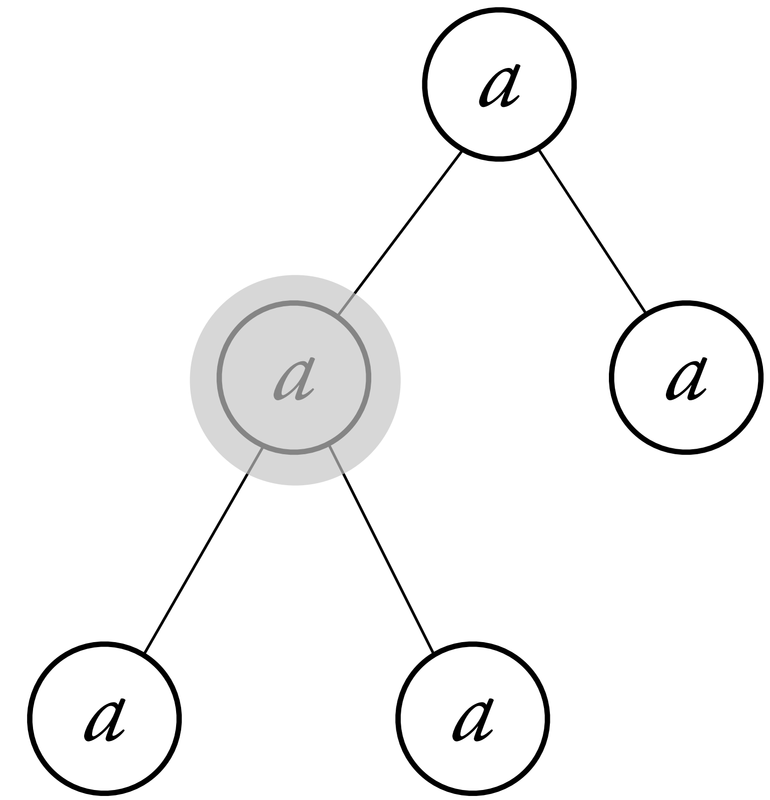
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

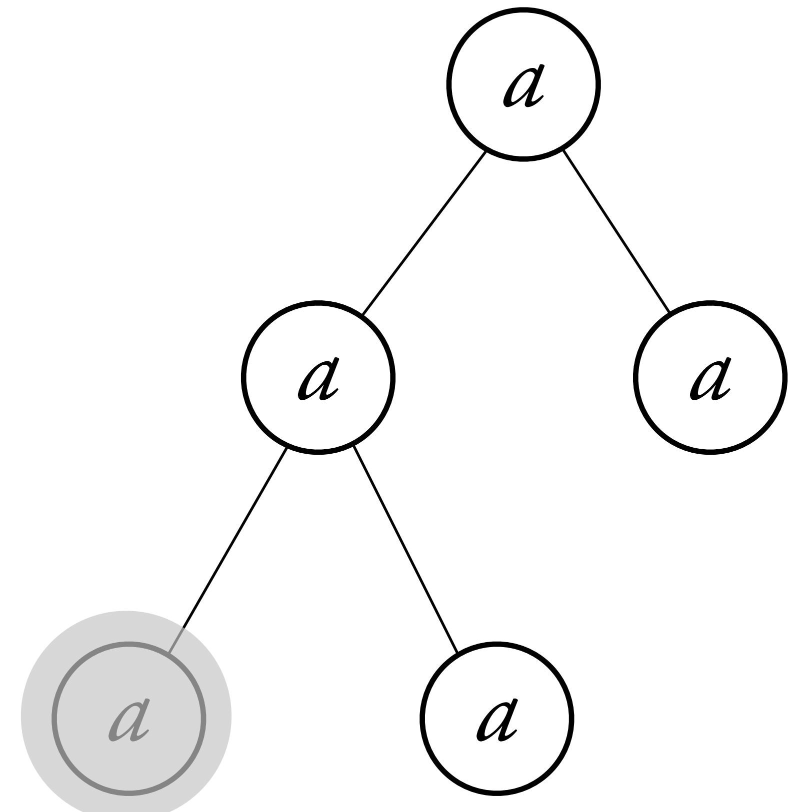
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

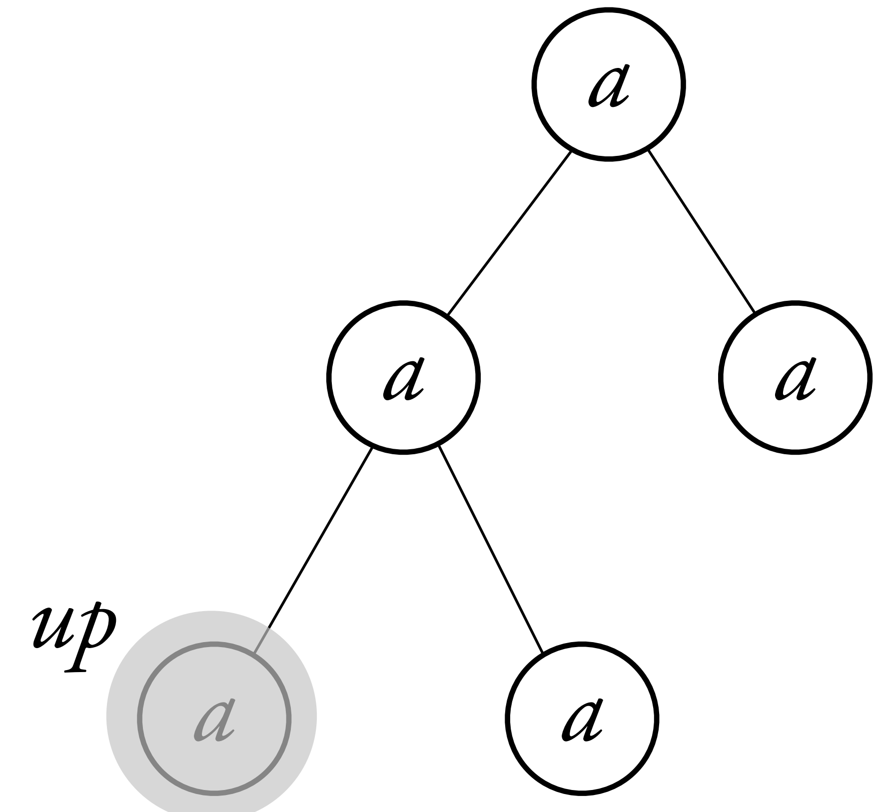
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label *b*

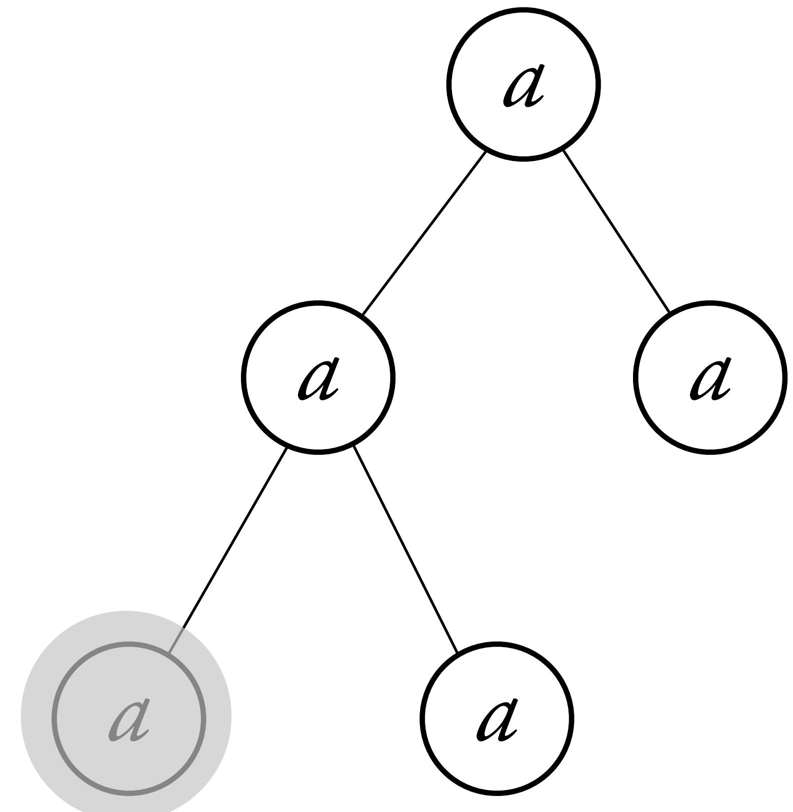
In state *p*, move left.
In state *p*, move right.
In state *p*, label *b*, accept.



Example.
All nodes have label *a*

Example.
Some node has label *b*

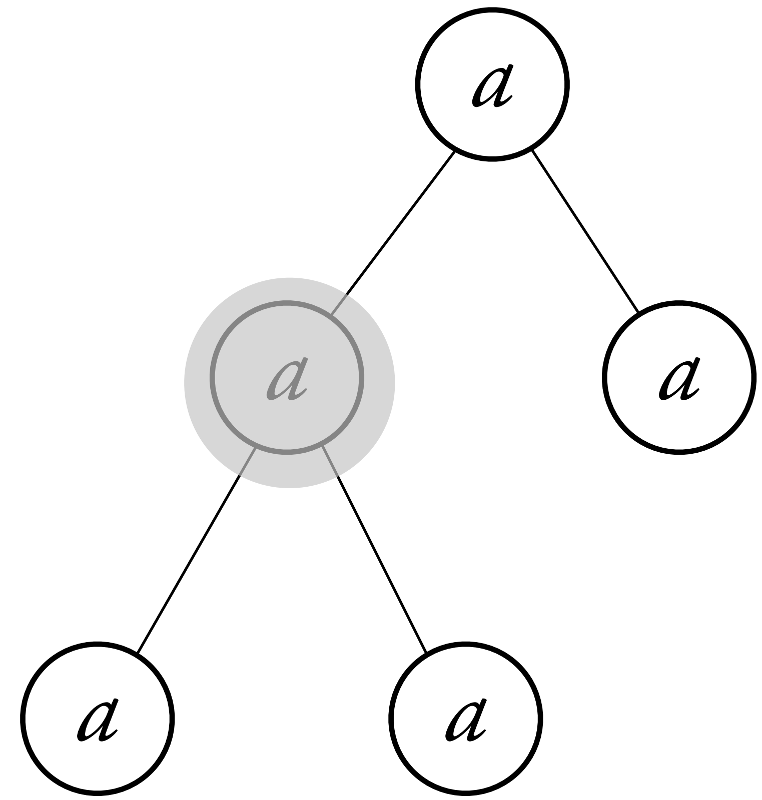
In state *p*, move left.
In state *p*, move right.
In state *p*, label *b*, accept.



Example.
All nodes have label *a*

Example.
Some node has label *b*

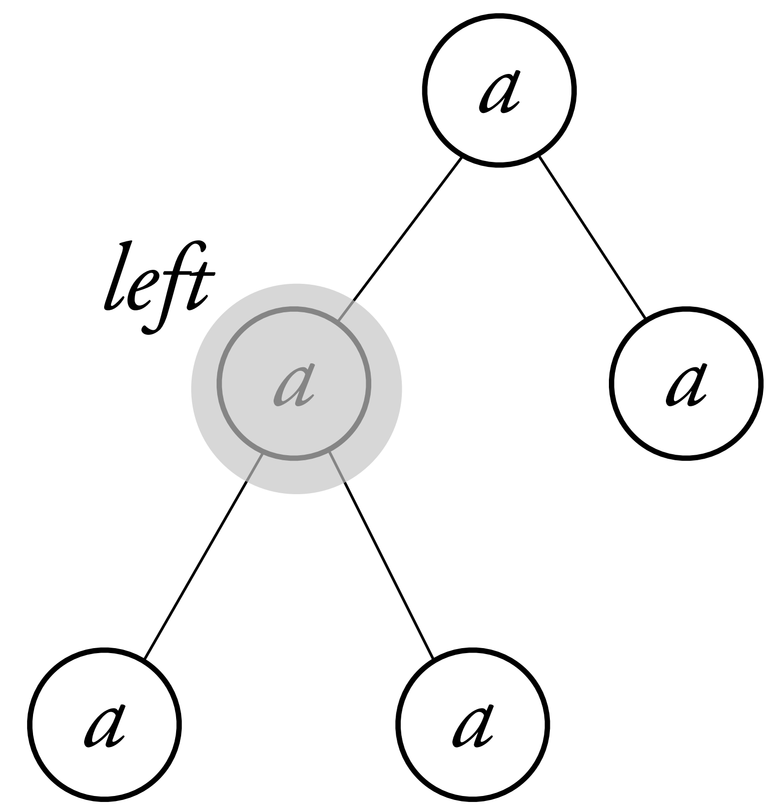
In state *p*, move left.
In state *p*, move right.
In state *p*, label *b*, accept.



Example.
All nodes have label *a*

Example.
Some node has label b

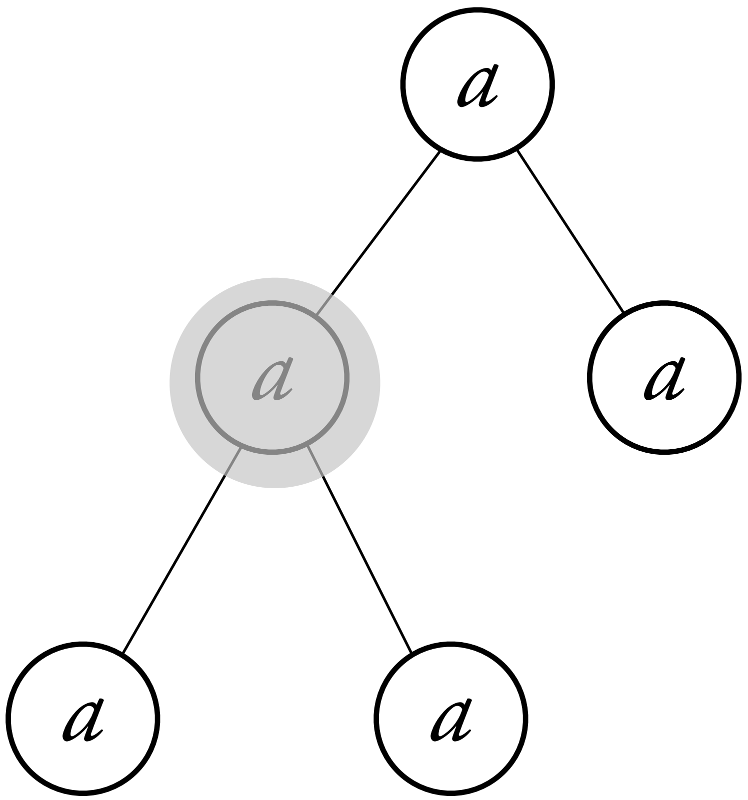
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

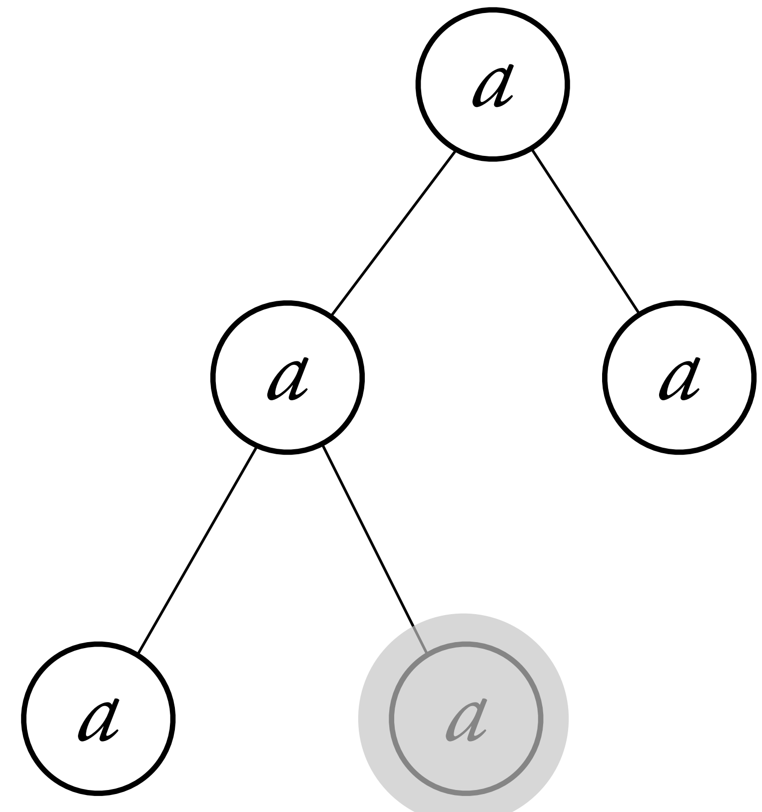
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

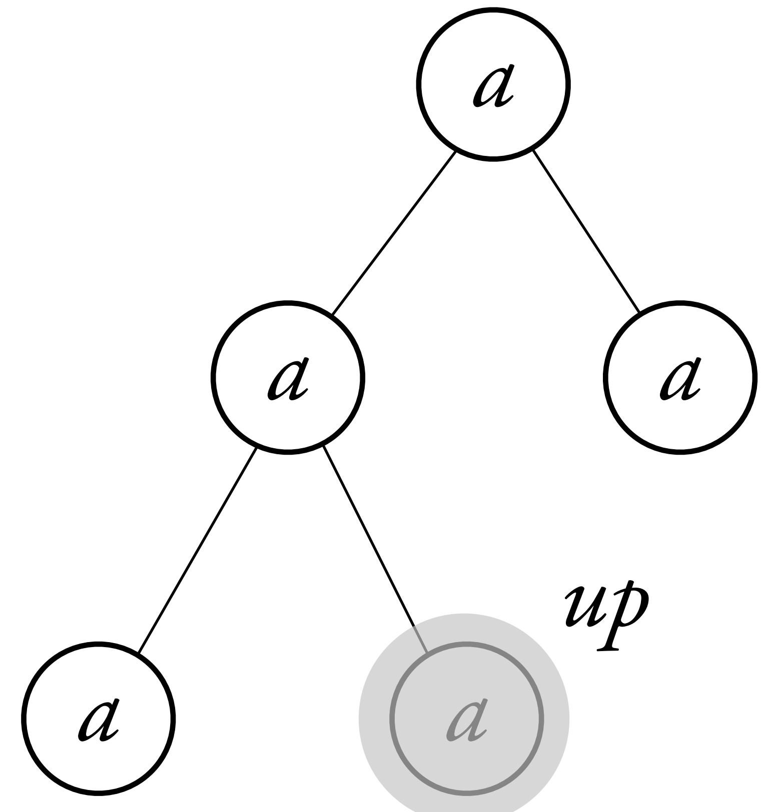
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

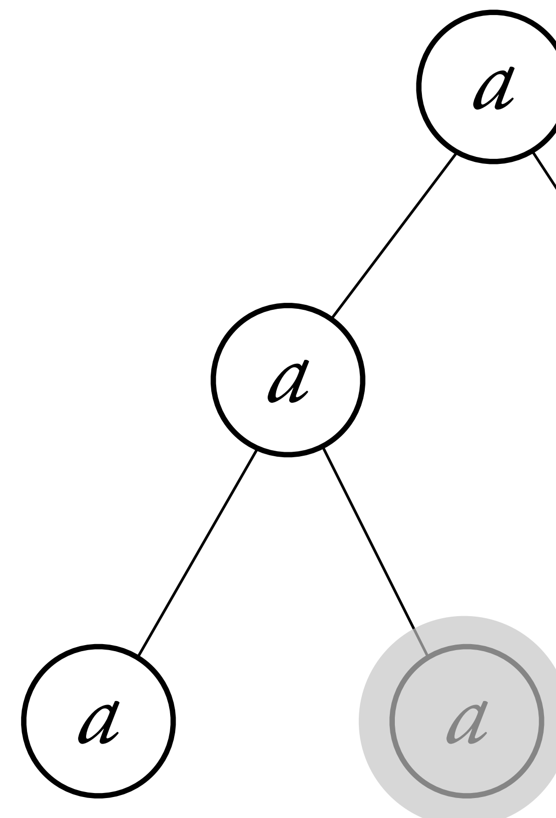
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label *b*

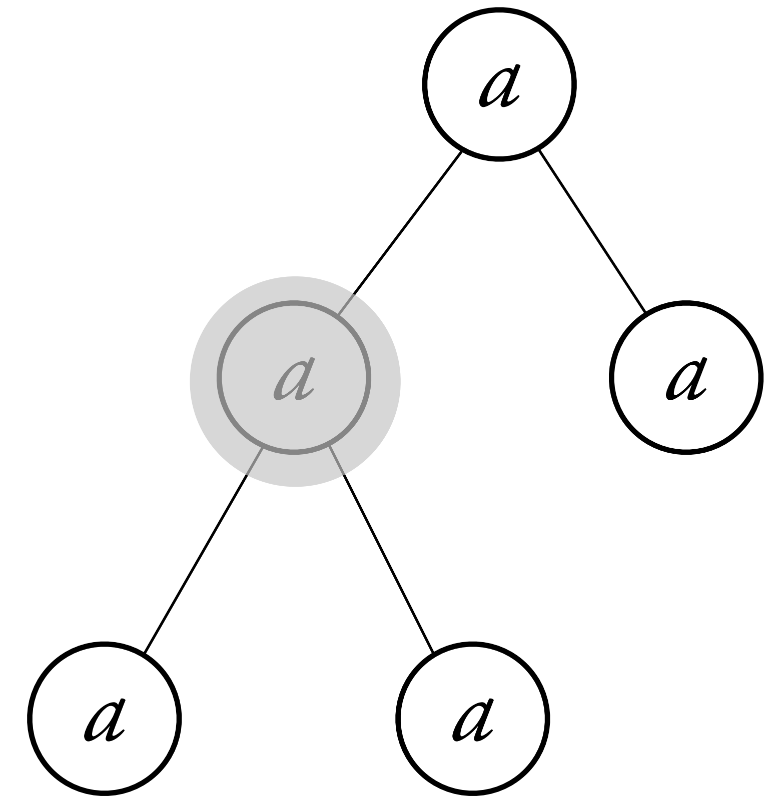
In state *p*, move left.
In state *p*, move right.
In state *p*, label *b*, accept.



Example.
All nodes have label *a*

Example.
Some node has label b

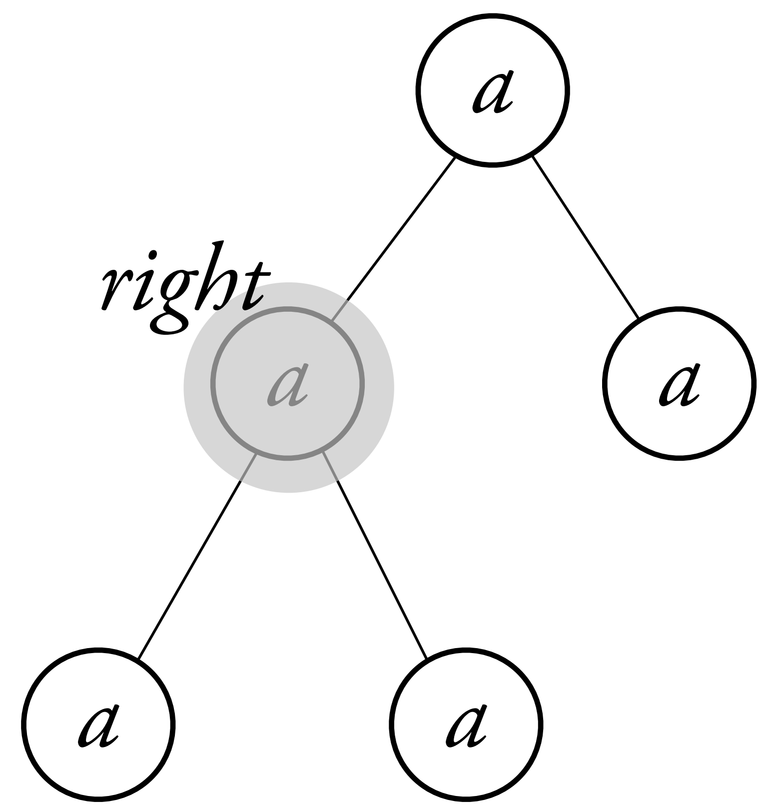
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

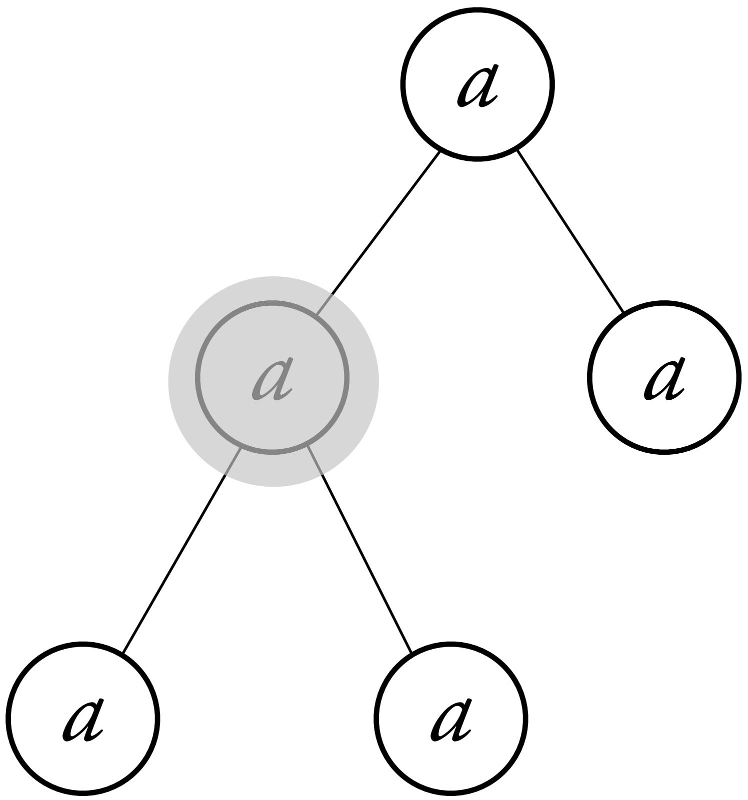
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

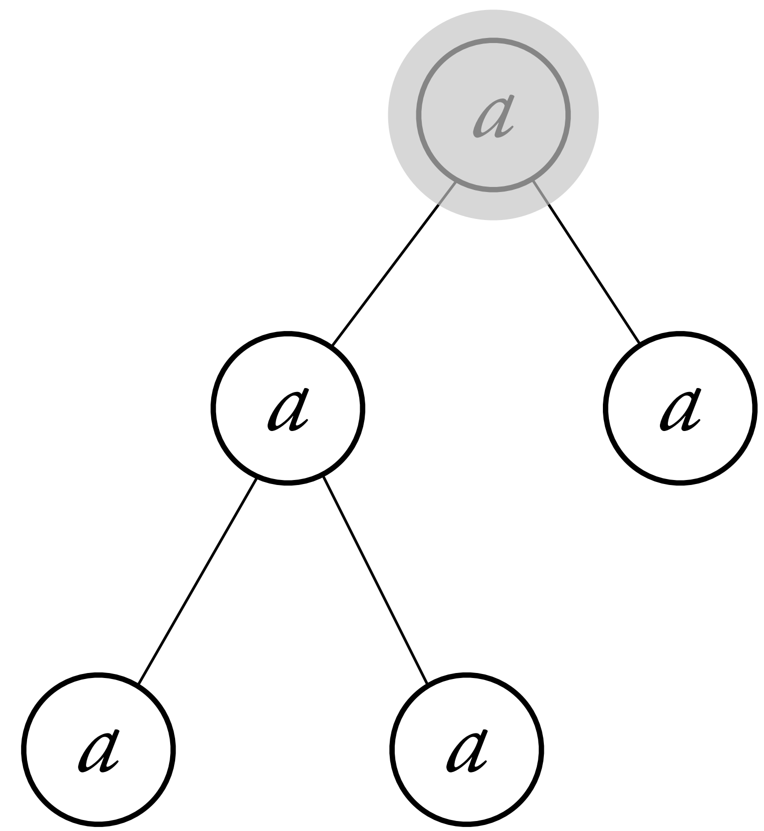
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

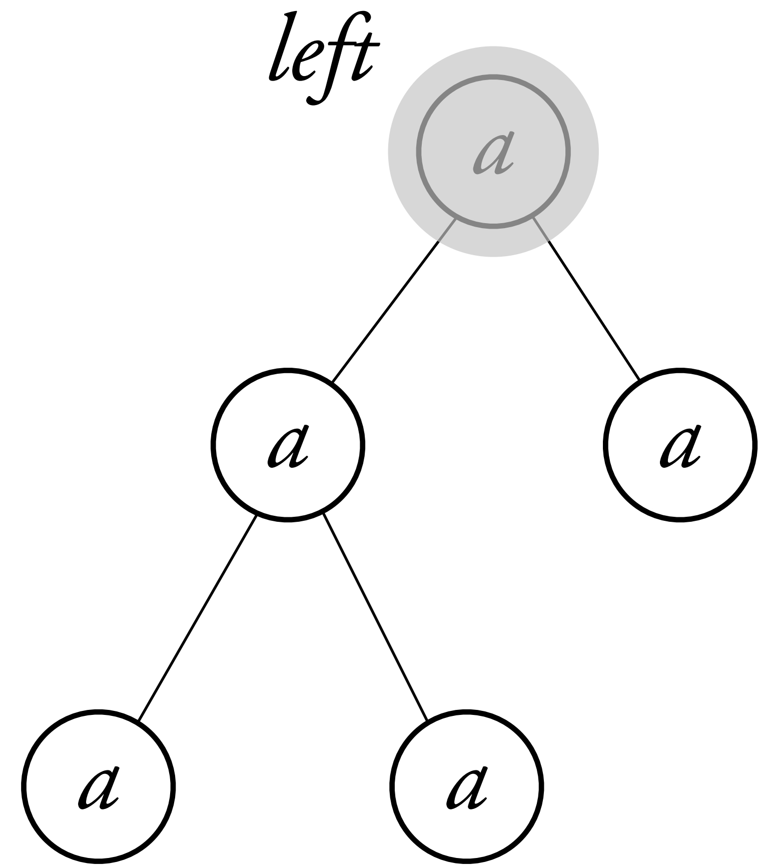
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

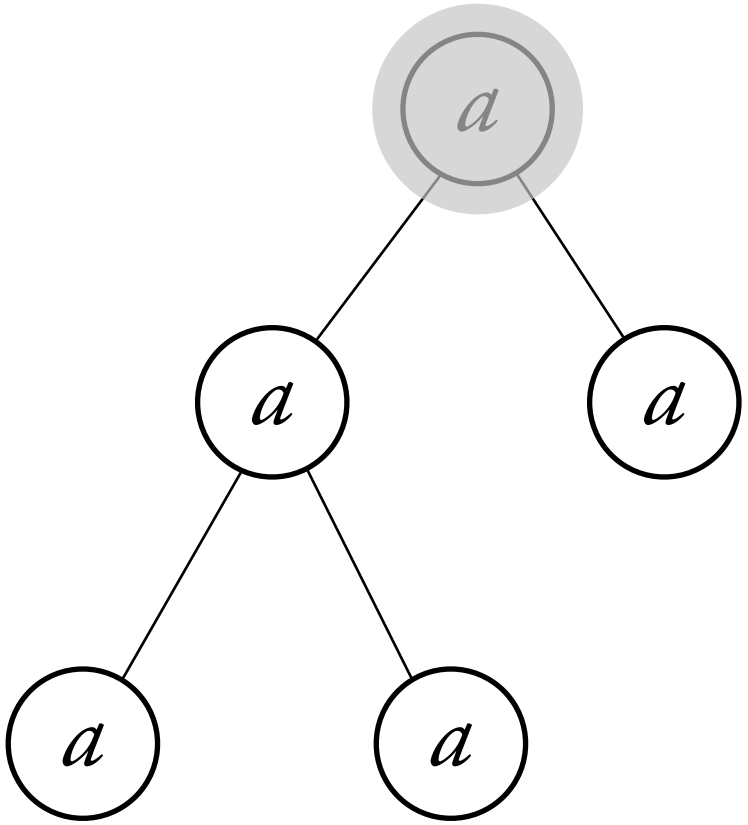
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

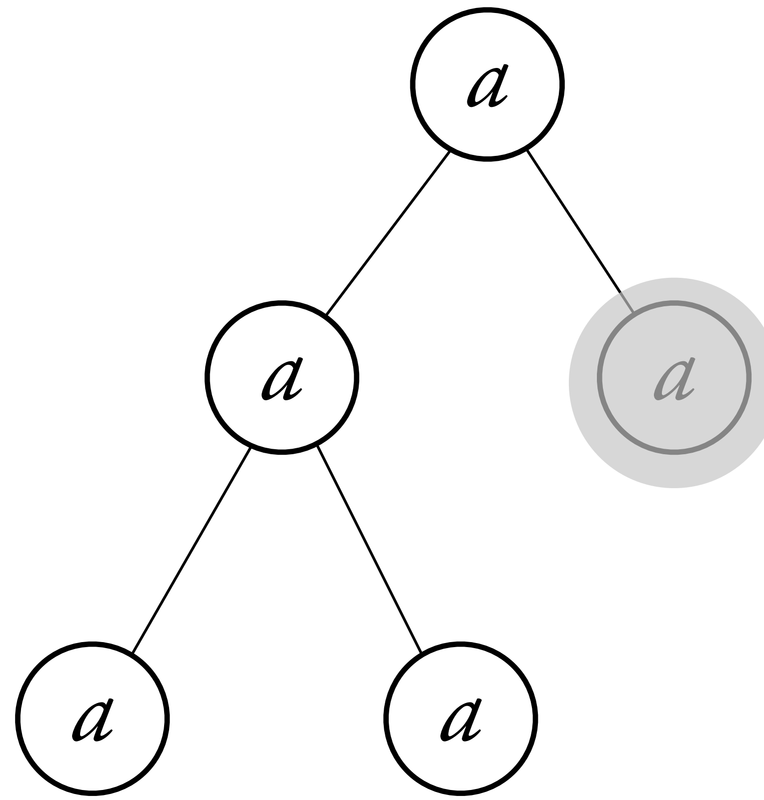
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

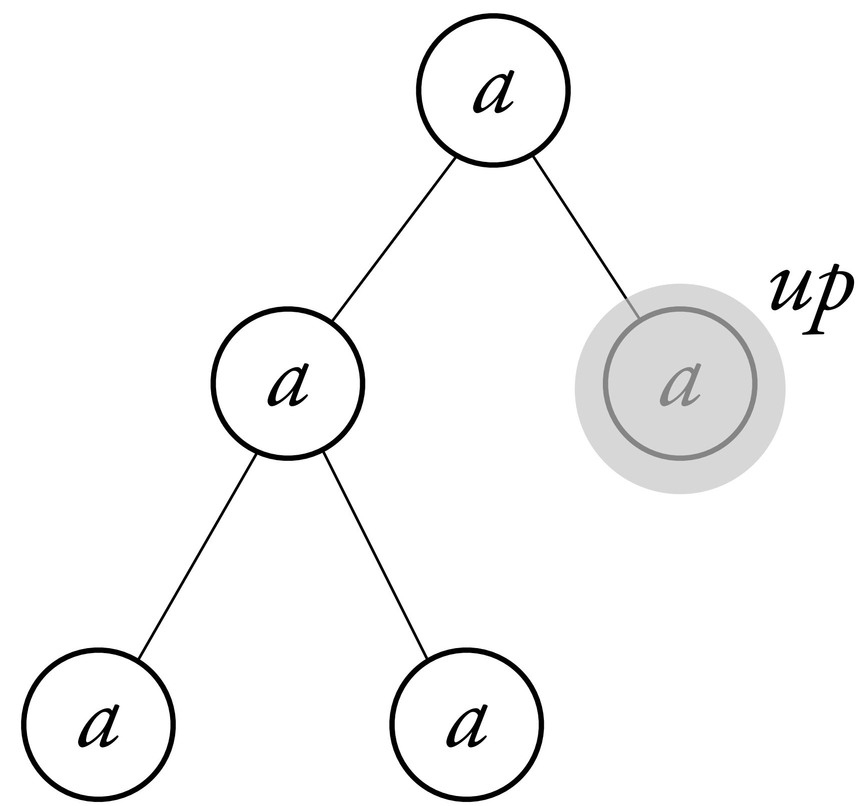
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label *b*

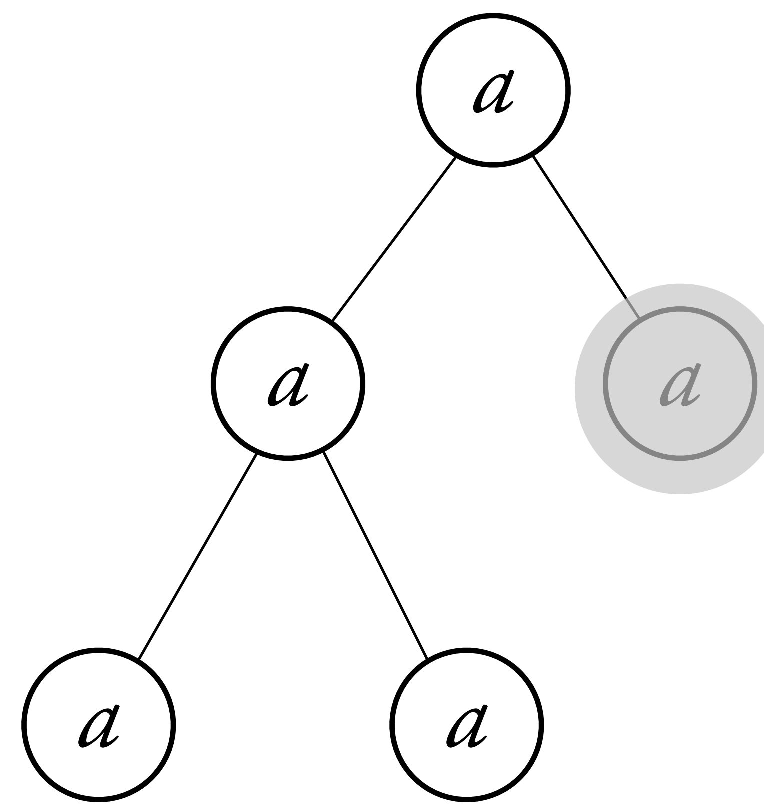
In state *p*, move left.
In state *p*, move right.
In state *p*, label *b*, accept.



Example.
All nodes have label *a*

Example.
Some node has label b

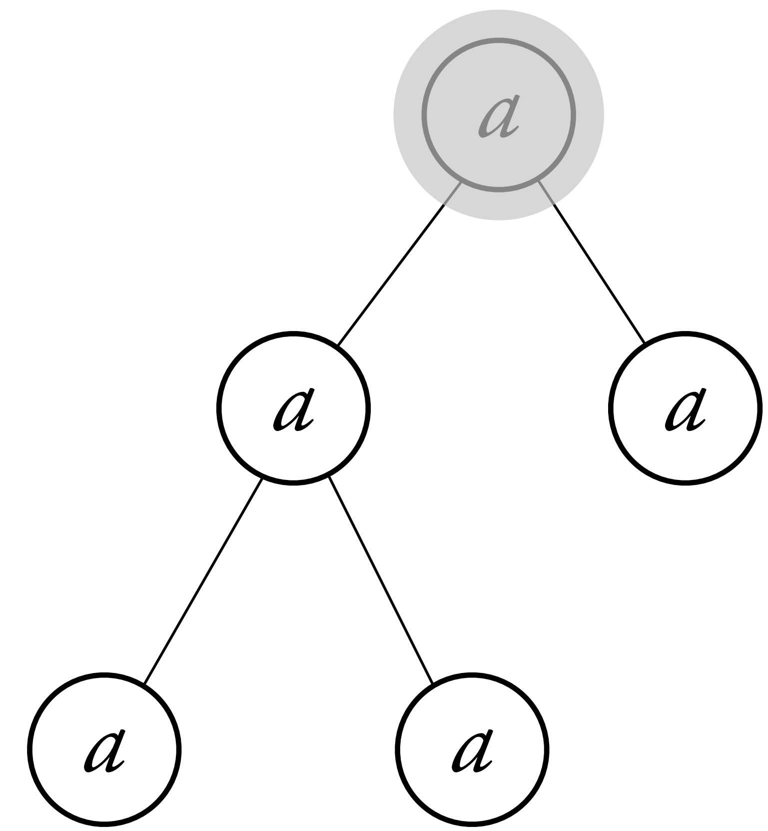
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

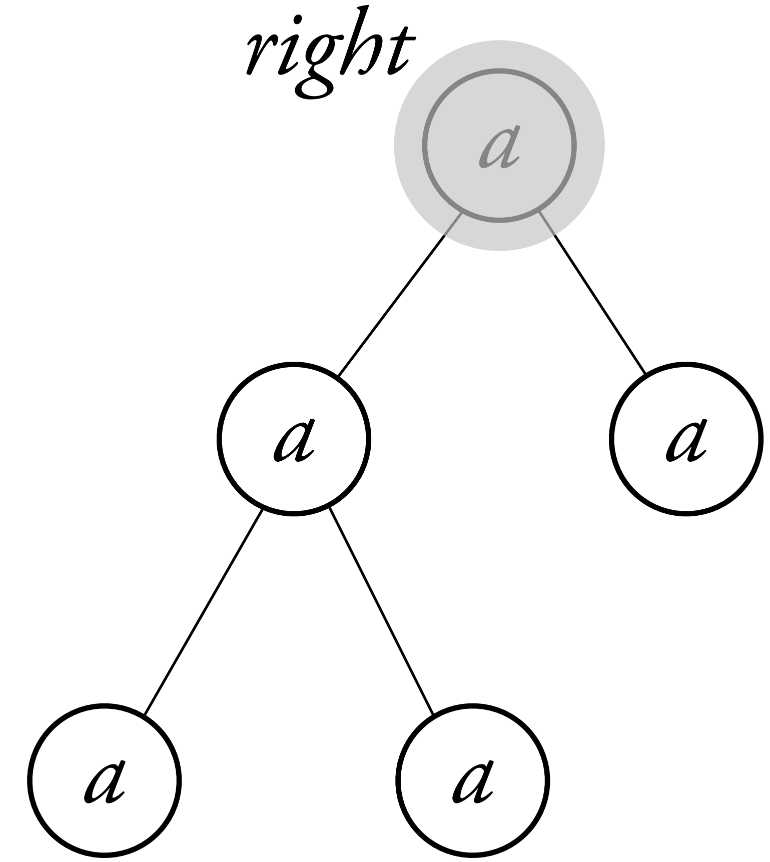
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

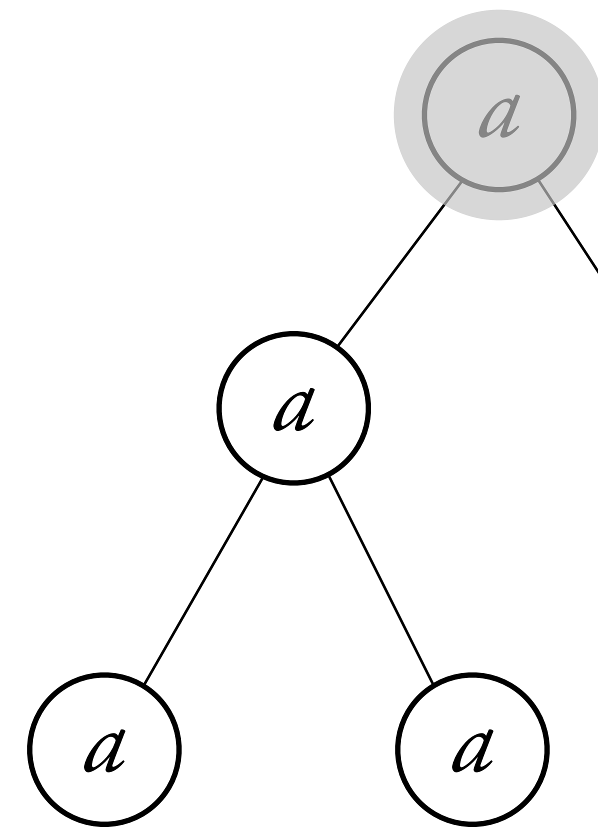
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

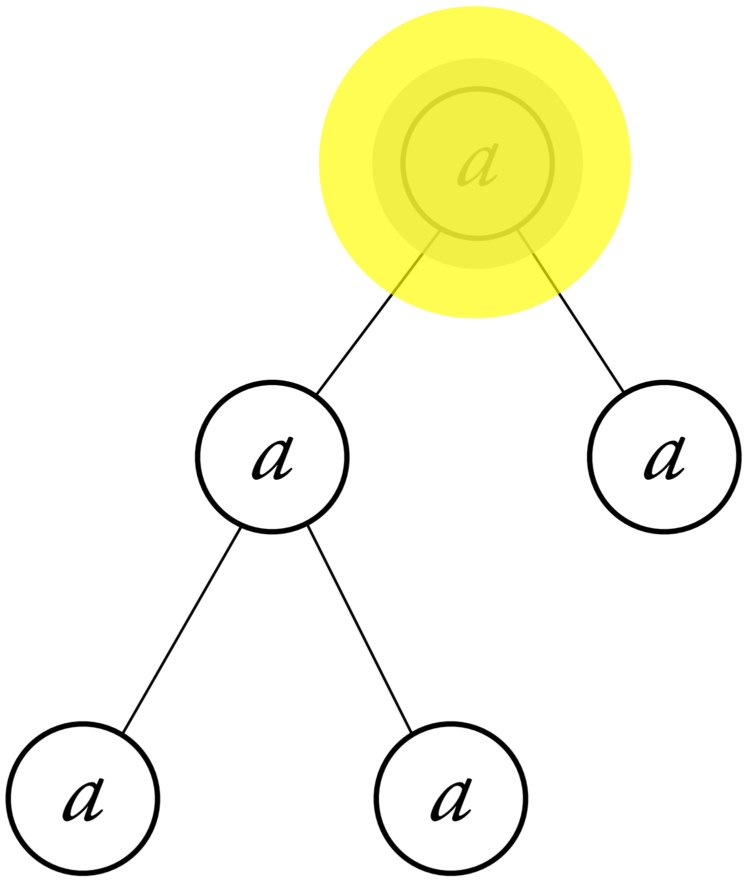
In state p, move left.
In state p, move right.
In state p, label b, accept.



Example.
All nodes have label a

Example.
Some node has label b

In state p, move left.
In state p, move right.
In state p, label b, accept.



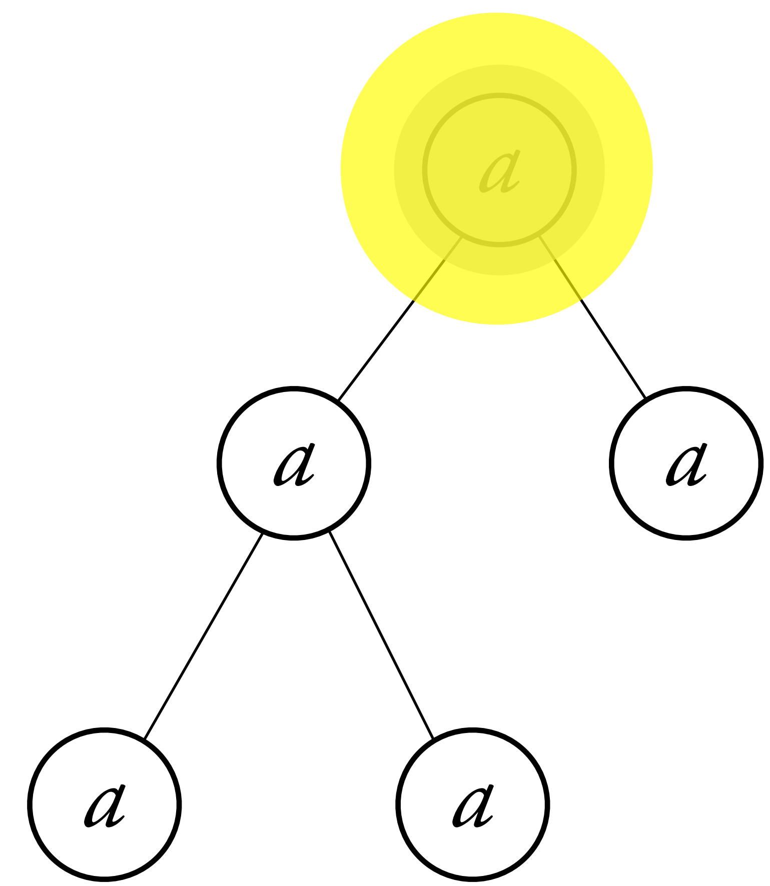
Example.
All nodes have label a

Example.
Some node has label b

In state p, move left.
In state p, move right.
In state p, label b, accept.

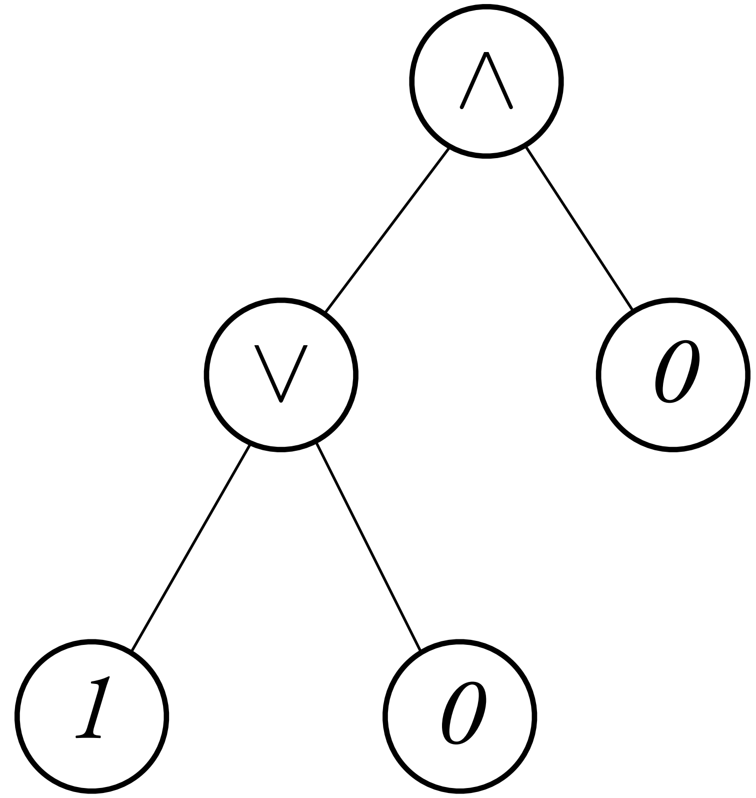
Complemenation is difficult!

Open problem:
Are nondeterministic tree-walking automata closed under complementation?

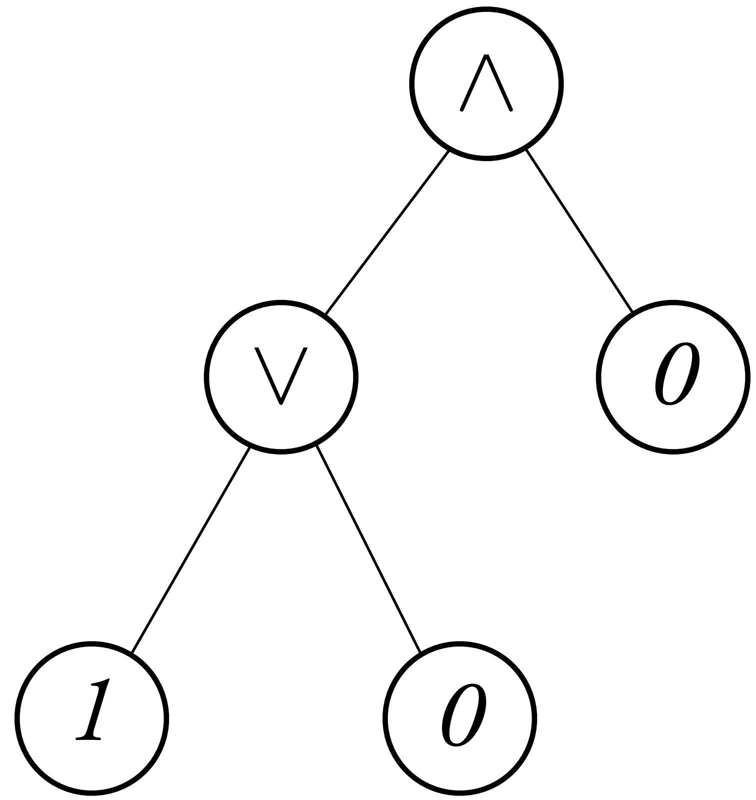


Example.
All nodes have label a

A clever tree-walking automaton



A clever tree-walking automaton

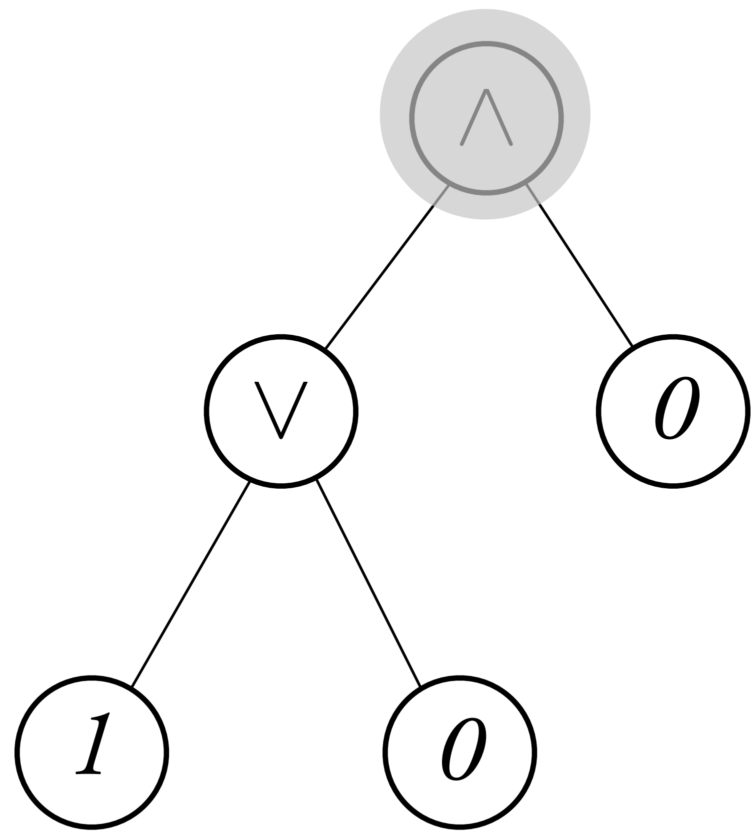


States: $\{q\} \cup \left(\underbrace{\{left, right\} \times \{0, 1\}}\right)$

first time

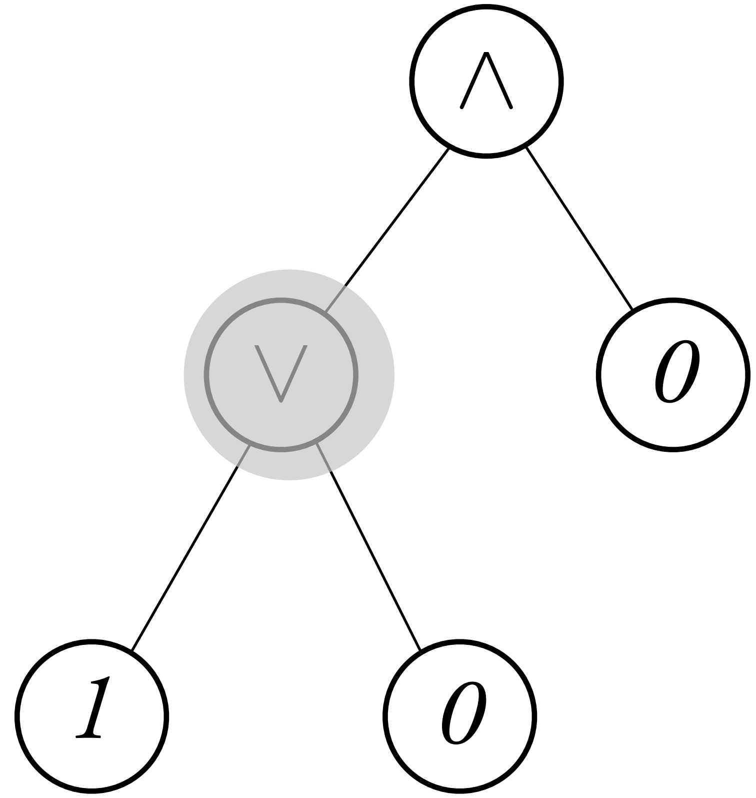
just evaluated evaluated left/right subtree to 0/1

A clever tree-walking automaton



$$\text{States: } \underbrace{\{q\}}_{\text{first time}} \cup \Big(\underbrace{\{left, right\} \times \{0,1\}}_{\text{just evaluated evaluated left/right subtree to 0/1}}\Big)$$

A clever tree-walking automaton

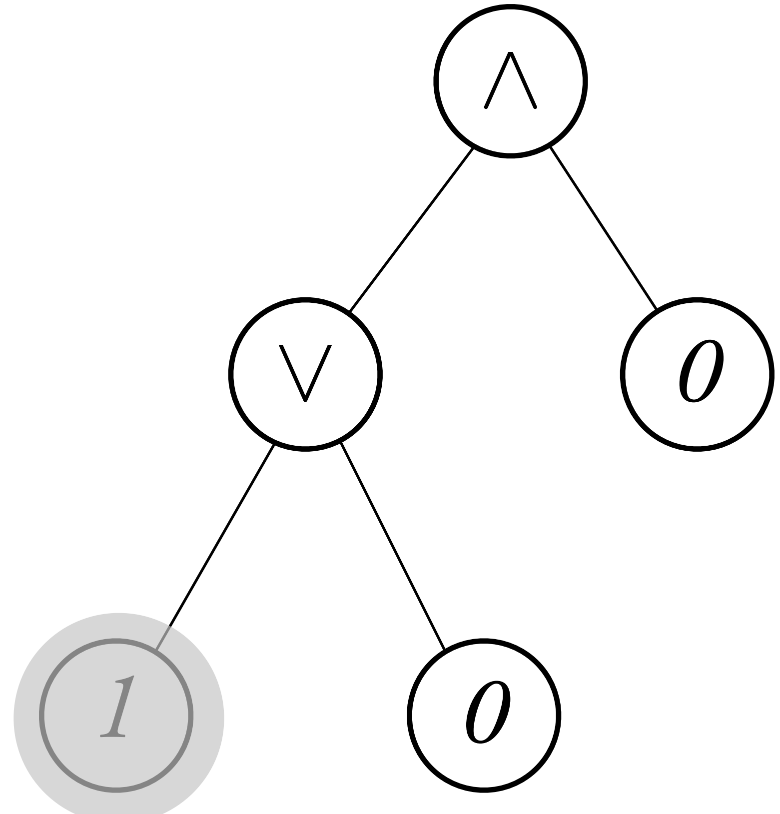


States: $\{q\} \cup \left(\underline{\{left, right\} \times \{0, 1\}}\right)$

first time

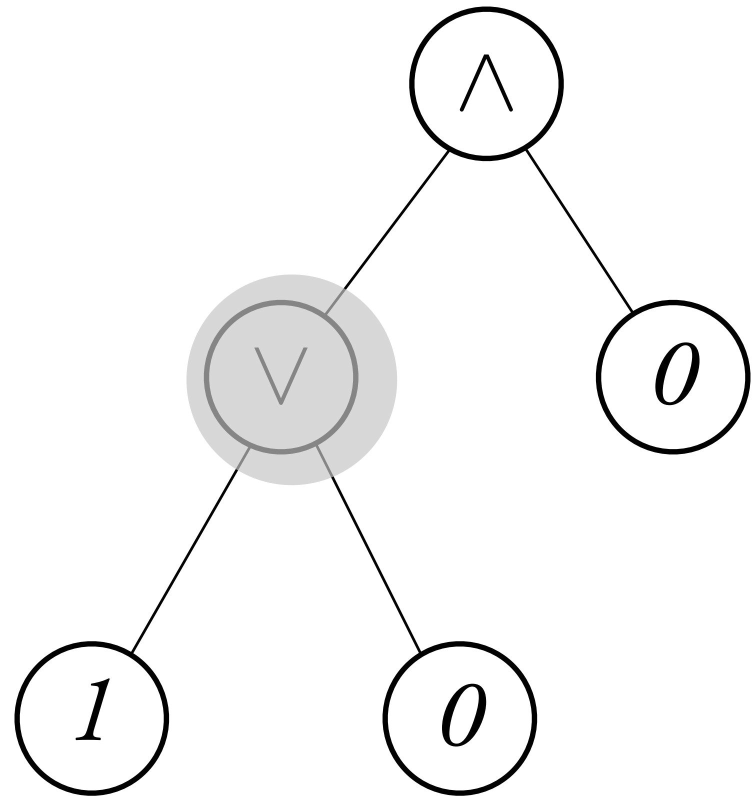
just evaluated evaluated left/right subtree to 0/1

A clever tree-walking automaton



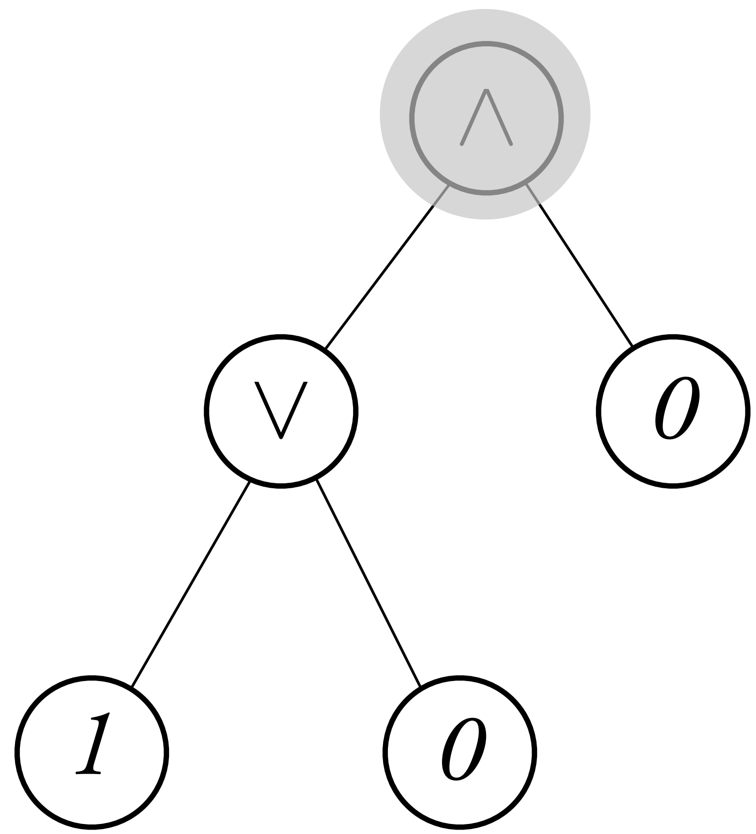
$$\text{States: } \underbrace{\{q\}}_{\text{first time}} \cup \Big(\underbrace{\{left, right\} \times \{0, 1\}}_{\text{just evaluated evaluated left/right subtree to 0/1}}\Big)$$

A clever tree-walking automaton



$$\text{States: } \underbrace{\{q\}}_{\text{first time}} \cup \Big(\underbrace{\{left, right\} \times \{0, 1\}}_{\text{just evaluated evaluated left/right subtree to 0/1}}\Big)$$

A clever tree-walking automaton

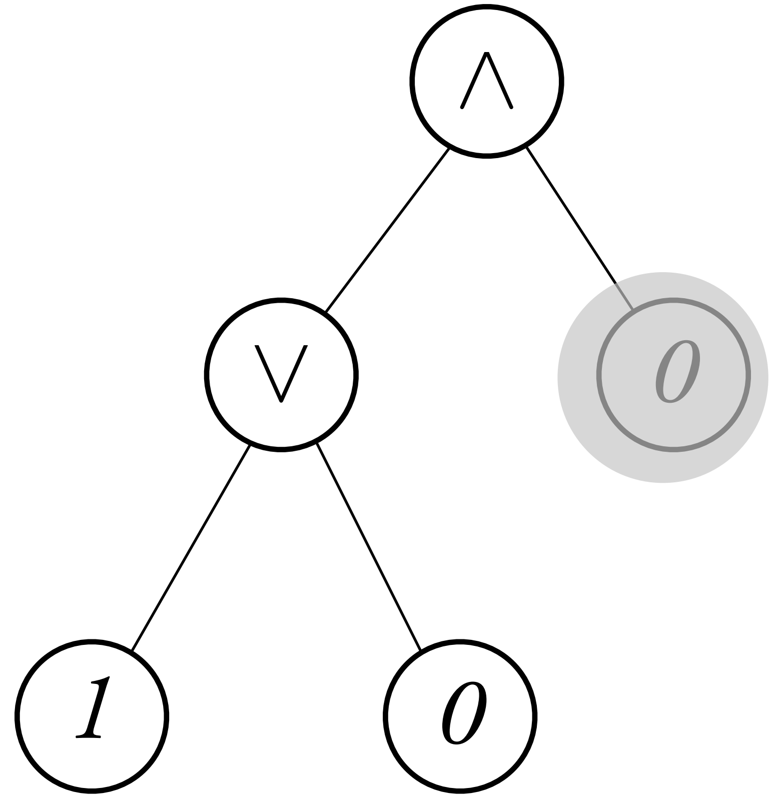


$$\text{States: } \{q\} \;\cup\; \big(\underbrace{\{left, right\} \times \{0, 1\}}_{}\big)$$

first time

just evaluated evaluated left/right subtree to 0/1

A clever tree-walking automaton

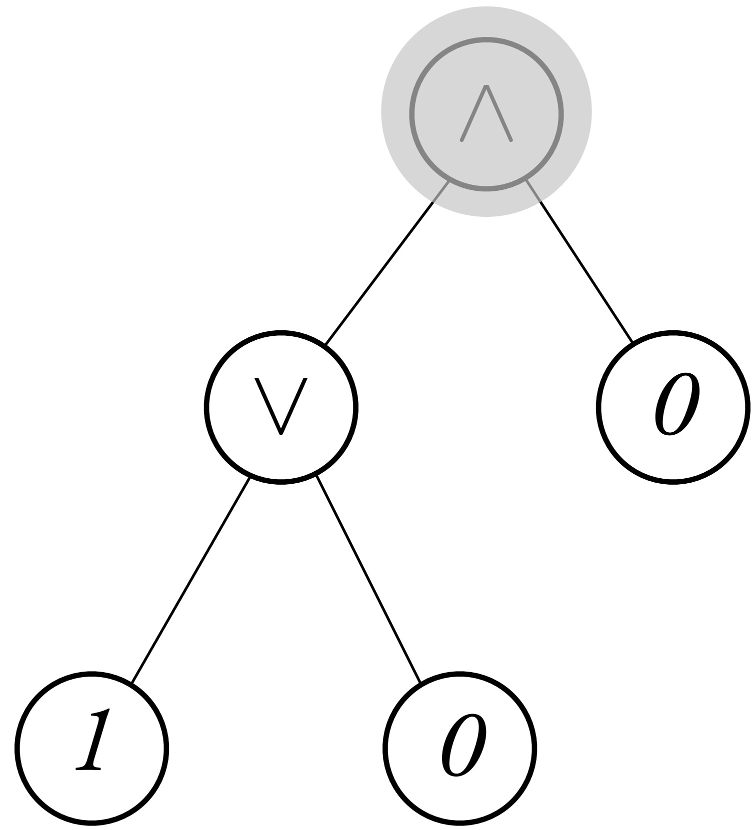


States: $\{q\} \cup \left(\underline{\{left, right\} \times \{0, 1\}}\right)$

first time

just evaluated evaluated left/right subtree to 0/1

A clever tree-walking automaton

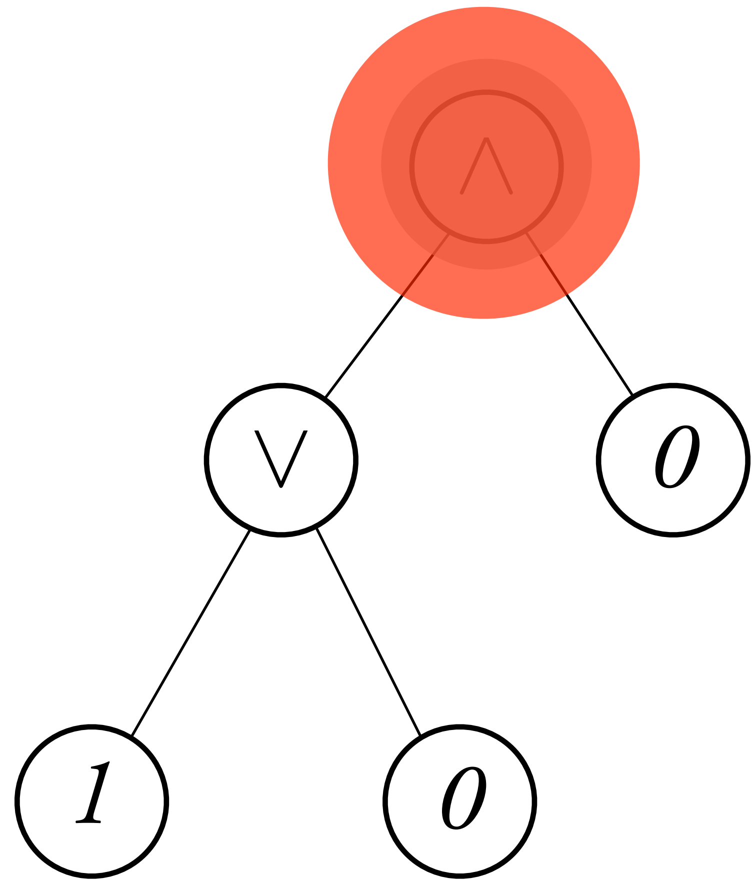


States: $\{q\} \;\cup\; \big(\underbrace{\{left, right\} \times \{0,1\}}\big)$

first time

just evaluated evaluated left/right subtree to 0/1

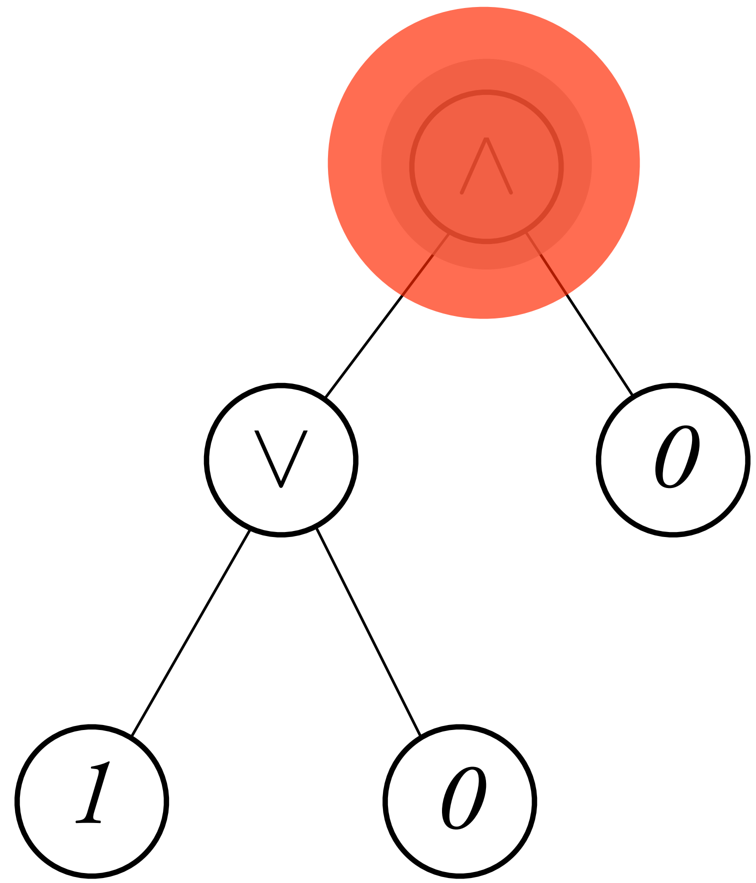
A clever tree-walking automaton



States: $\underbrace{\{q\}}_{\text{first time}} \cup \left(\underbrace{\{left, right\} \times \{0, 1\}}_{\text{just evaluated evaluated left/right subtree to 0/1}}\right)$

A clever tree-walking automaton

still works with negation, but what about XOR?



States: $\{q\} \cup (\underbrace{\{left, right\} \times \{0, 1\}}_{\text{just evaluated evaluated left/right subtree to 0/1}})$

first time

just evaluated evaluated left/right subtree to 0/1

Complemenation is difficult!

Open problem:
Are nondeterministic tree-walking automata closed under complementation?

Complemenation is difficult!

Open problem:
Are nondeterministic tree-walking automata closed under complementation?

Theorem. [Muscholl, Samuelides, Segoufin]
Deterministic tree-walking automata are closed under complementation.

Complemenation is difficult!

Open problem:
Are nondeterministic tree-walking automata closed under complementation?

Theorem. [Muscholl, Samuelides, Segoufin]
Deterministic tree-walking automata are closed under complementation.

Lemma.
Every deterministic tree-walking automaton is equivalent to one that ends every run with a *reject* or *accept* command.

Plan

-A tree-walking automaton

-Expressive power

-Pebble automata and logic

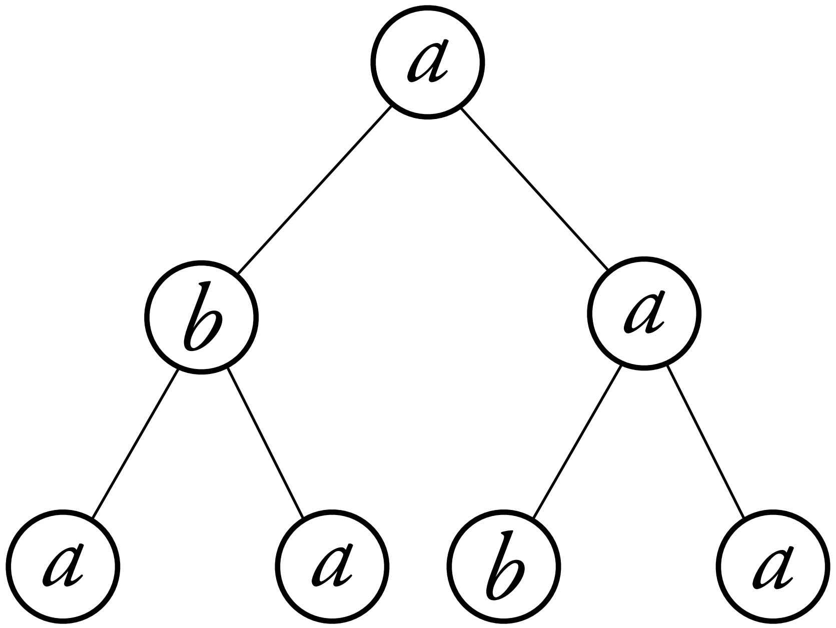
Plan

-A tree-walking automaton

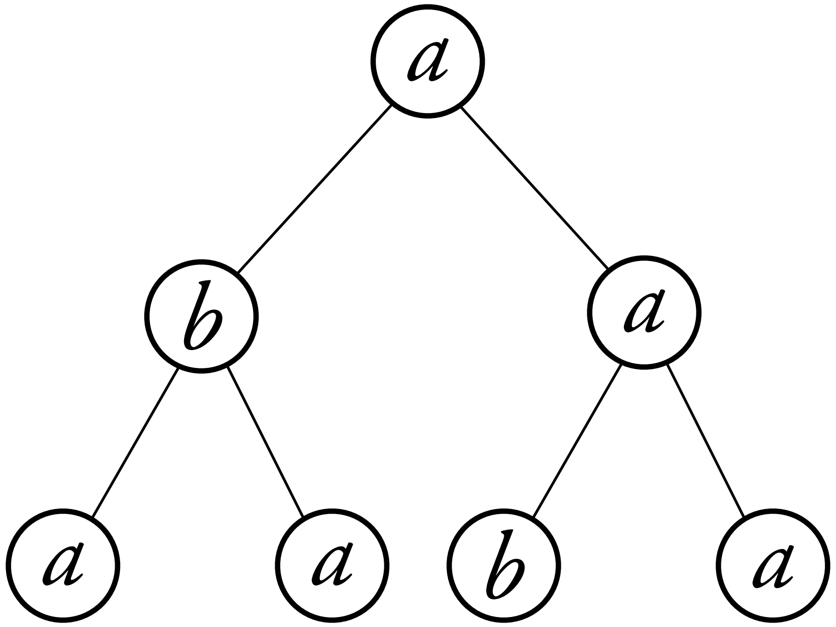
-Expressive power

-Pebble automata and logic

How do tree-walking automata relate to “real” tree automata?



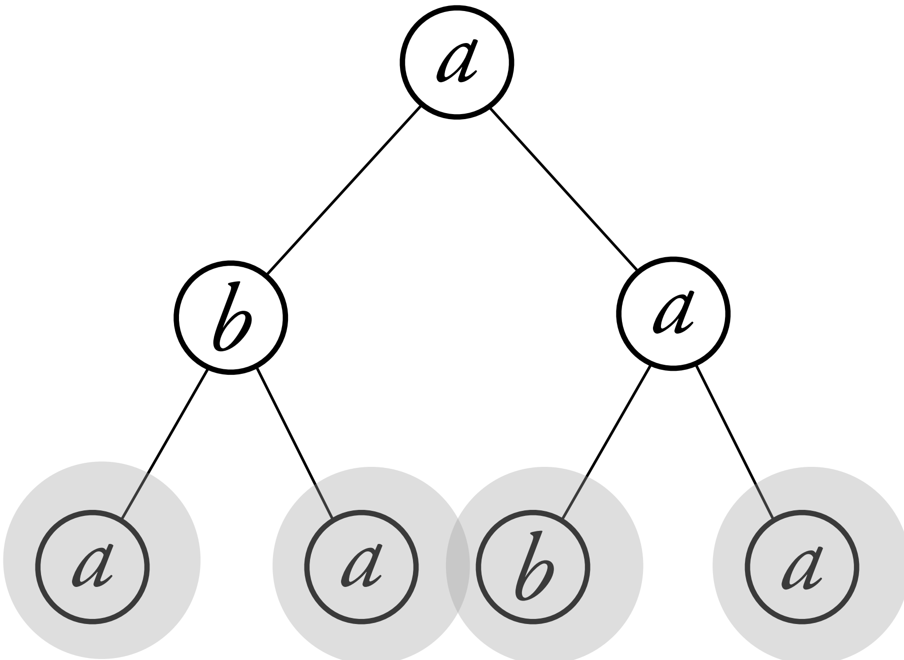
How do tree-walking automata relate to “real” tree automata?



Standard model is a *branching automaton*.

Here we use bottom-up deterministic branching automata.

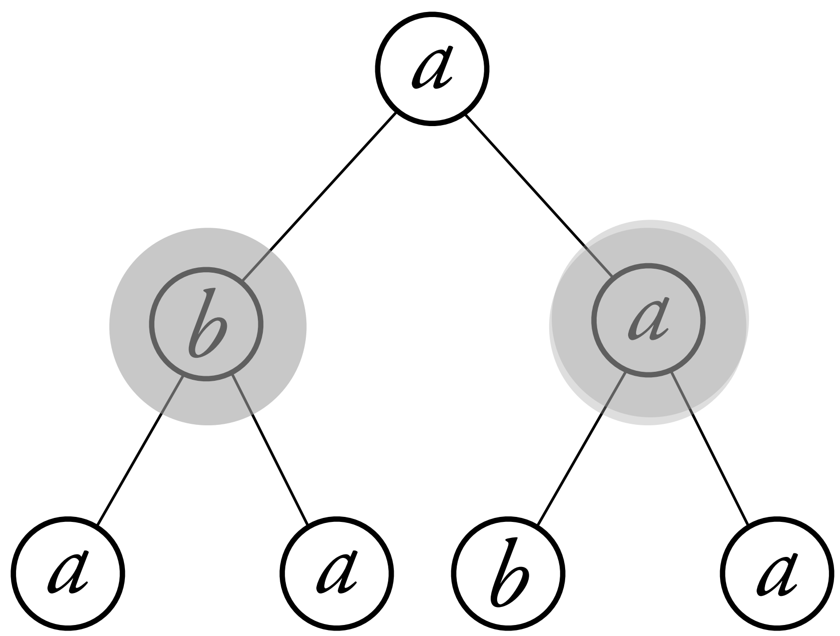
How do tree-walking automata relate to “real” tree automata?



Standard model is a *branching automaton*.

Here we use bottom-up deterministic branching automata.

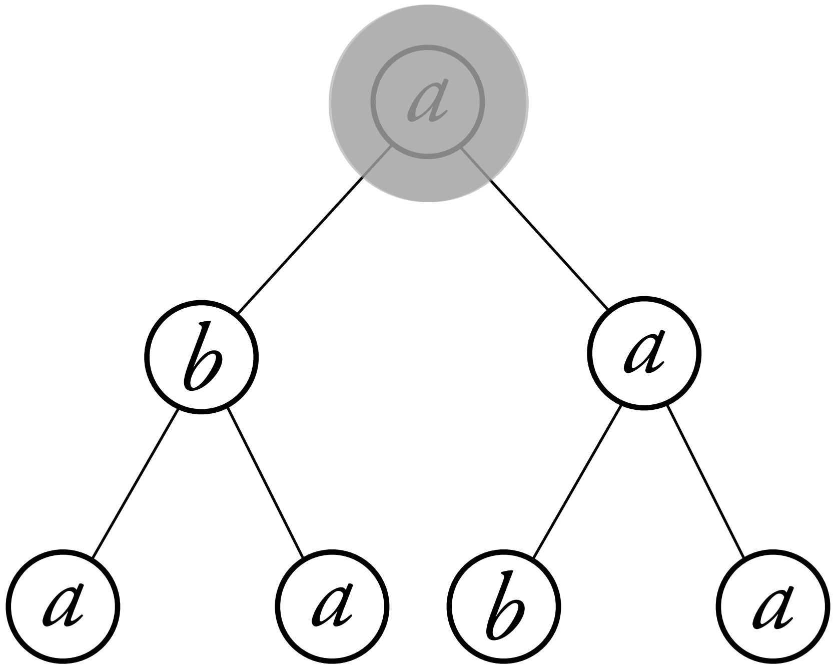
How do tree-walking automata relate to “real” tree automata?



Standard model is a *branching automaton.*

Here we use bottom-up deterministic branching automata.

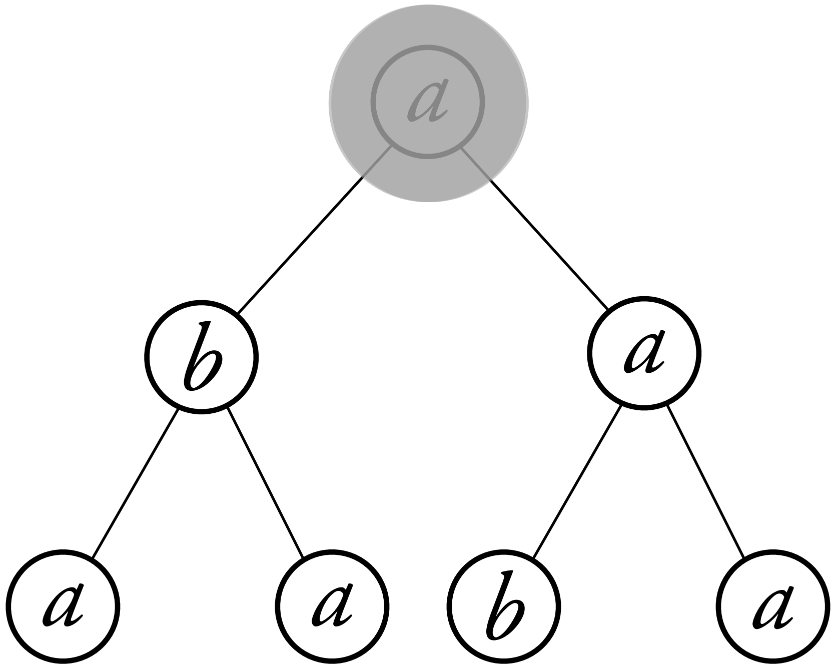
How do tree-walking automata relate to “real” tree automata?



Standard model is a *branching automaton.*

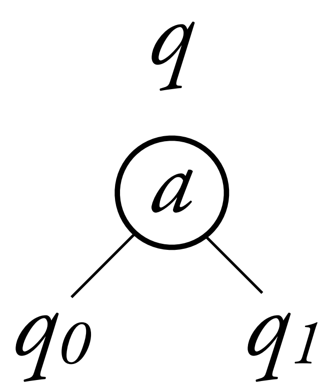
Here we use bottom-up deterministic branching automata.

How do tree-walking automata relate to “real” tree automata?



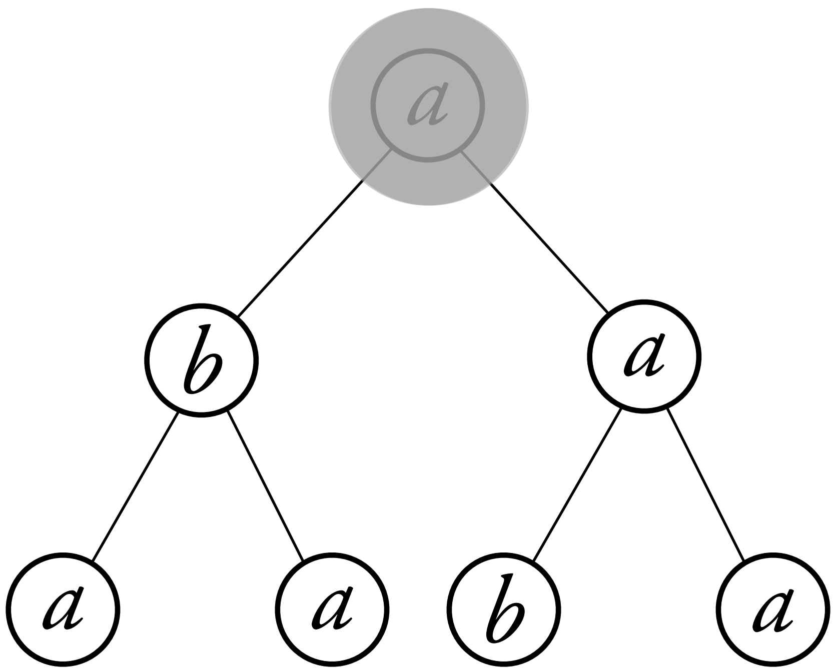
Standard model is a *branching automaton*.

Here we use bottom-up deterministic branching automata.



If the root label is *a*, the left subtree has value q_0 , and the right subtree has value q_1 , then the whole tree has value q.

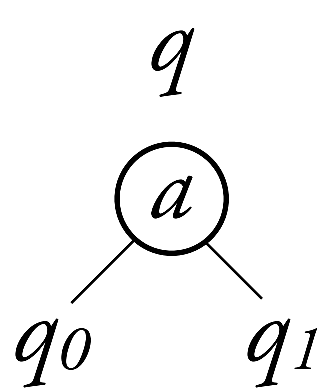
How do tree-walking automata relate to "real" tree automata?



Standard model is a *branching automaton.*

Here we use bottom-up deterministic branching automata.

$$\langle Q, q_I, \Sigma, \delta, F \rangle$$



If the root label is a, the left subtree has value q_0, and the right subtree has value q_1, then the whole tree has value q.

Branching automata are closed under union, intersection, complementation, projection etc.

Def. A tree language is called *regular* if it is recognized by a branching automaton.

Question: how do tree-walking automata relate to regular languages?

$$\mathrm{TWA} \subseteq \mathrm{REG}$$

Question: how do tree-walking automata relate to regular languages?

$$\text{TWA} \subseteq \text{REG}$$

To a tree-walking automaton

$$\langle Q, q_I, \Sigma, \delta \rangle$$

we associate a branching automaton that accepts the same trees.

States $P(Q \times Q)$

Question: how do tree-walking automata relate to regular languages?

$$\mathrm{TWA} \subseteq \mathrm{REG}$$

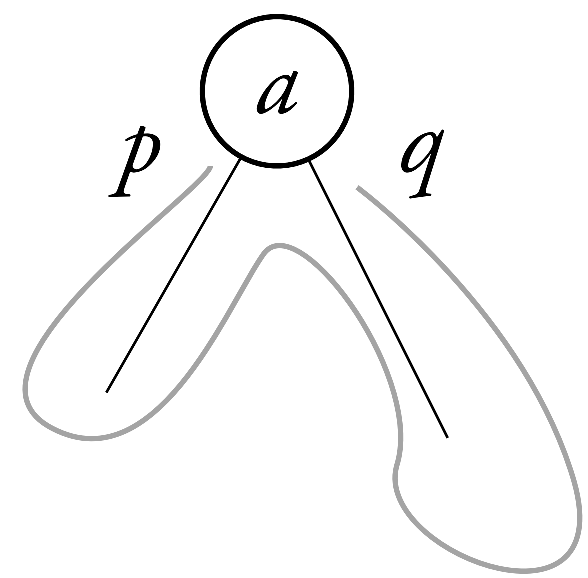
To a tree-walking automaton

$$\langle Q, q_I, \Sigma, \delta \rangle$$

we associate a branching automaton that accepts the same trees.

States $P(Q \times Q)$

Value of a tree: set of pairs *(p,q)* that give a loop in the root:



(these are loops that stay below the root)

Question: how do tree-walking automata relate to regular languages?

$$\mathrm{TWA} \subseteq \mathrm{REG}$$

To a tree-walking automaton

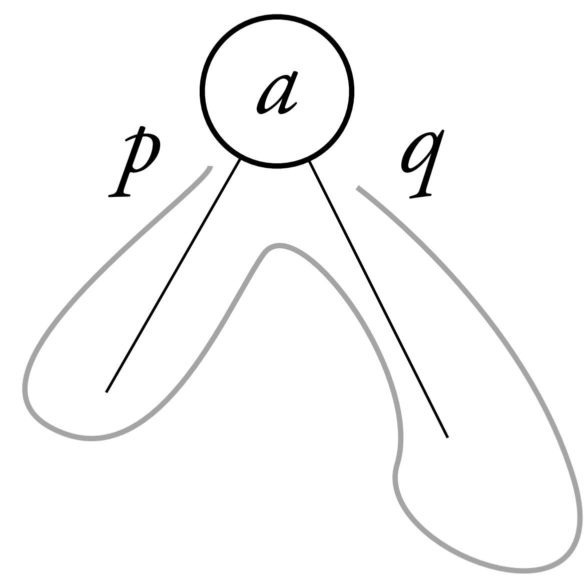
$$\langle Q, q_I, \Sigma, \delta \rangle$$

we associate a branching automaton that accepts the same trees.

States $\cancel{P(Q \times Q)}$

$$P(Q \times Q \times \{left, right, root\})$$

Value of a tree: set of pairs *(p,q)* that give a loop in the root:



(these are loops that stay below the root)

Question: how do tree-walking automata relate to regular languages?

$$\text{TWA} \subseteq \text{REG}$$

To a tree-walking automaton

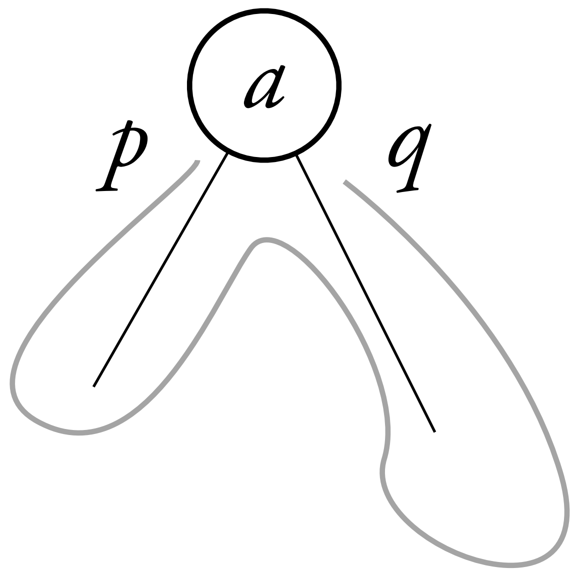
$$\langle Q, q_I, \Sigma, \delta \rangle$$

we associate a branching automaton that accepts the same trees.

States $\cancel{P(Q \times Q)}$

$$P(Q \times Q \times \{left, right, root\})$$

Value of a tree: set of pairs *(p,q)* that give a loop in the root:



(these are loops that stay below the root)

Corollary. Emptiness for tree-walking automata is in EXPTIME.

Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

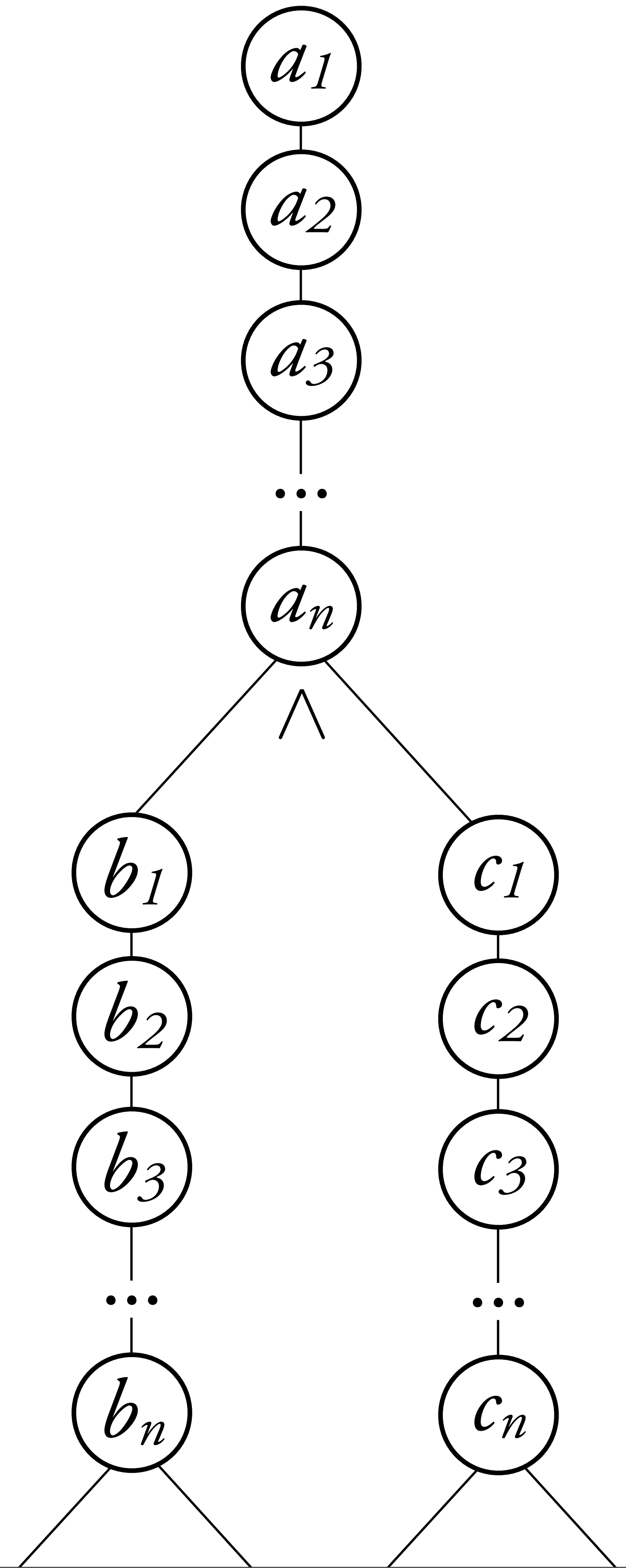
The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

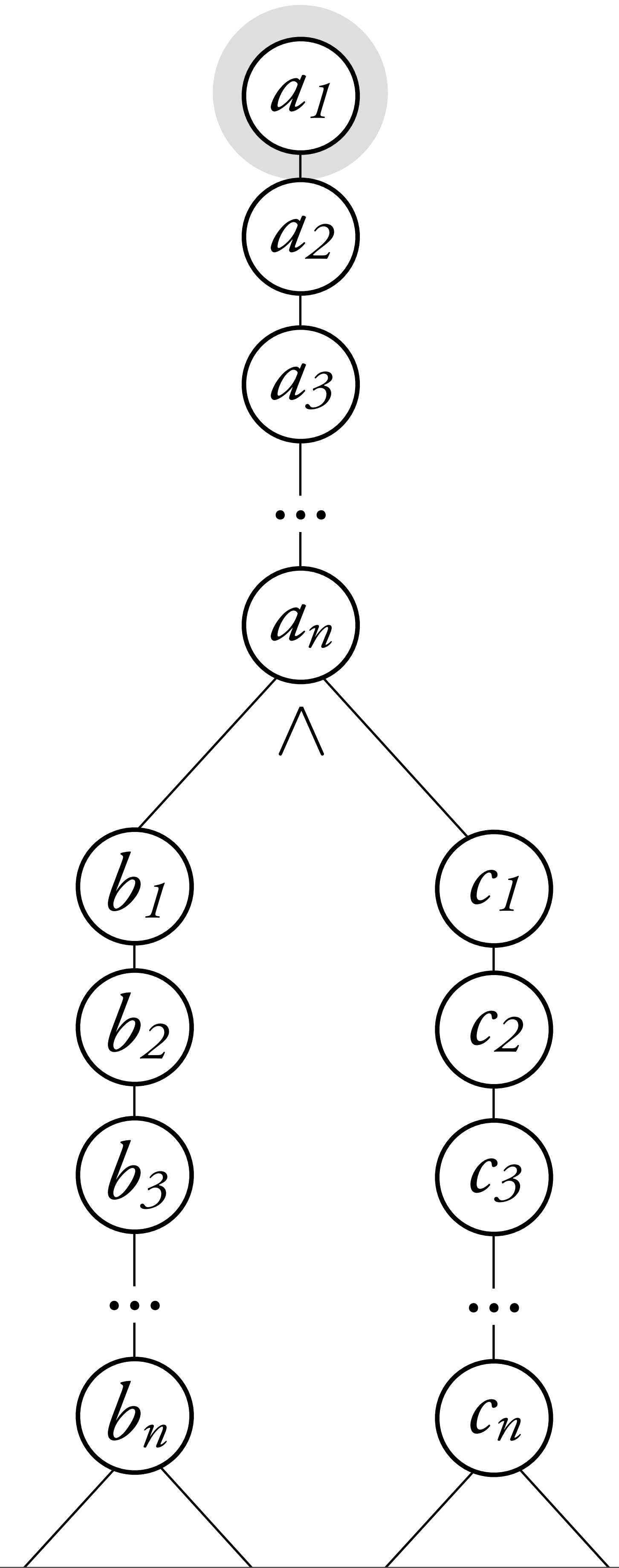


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

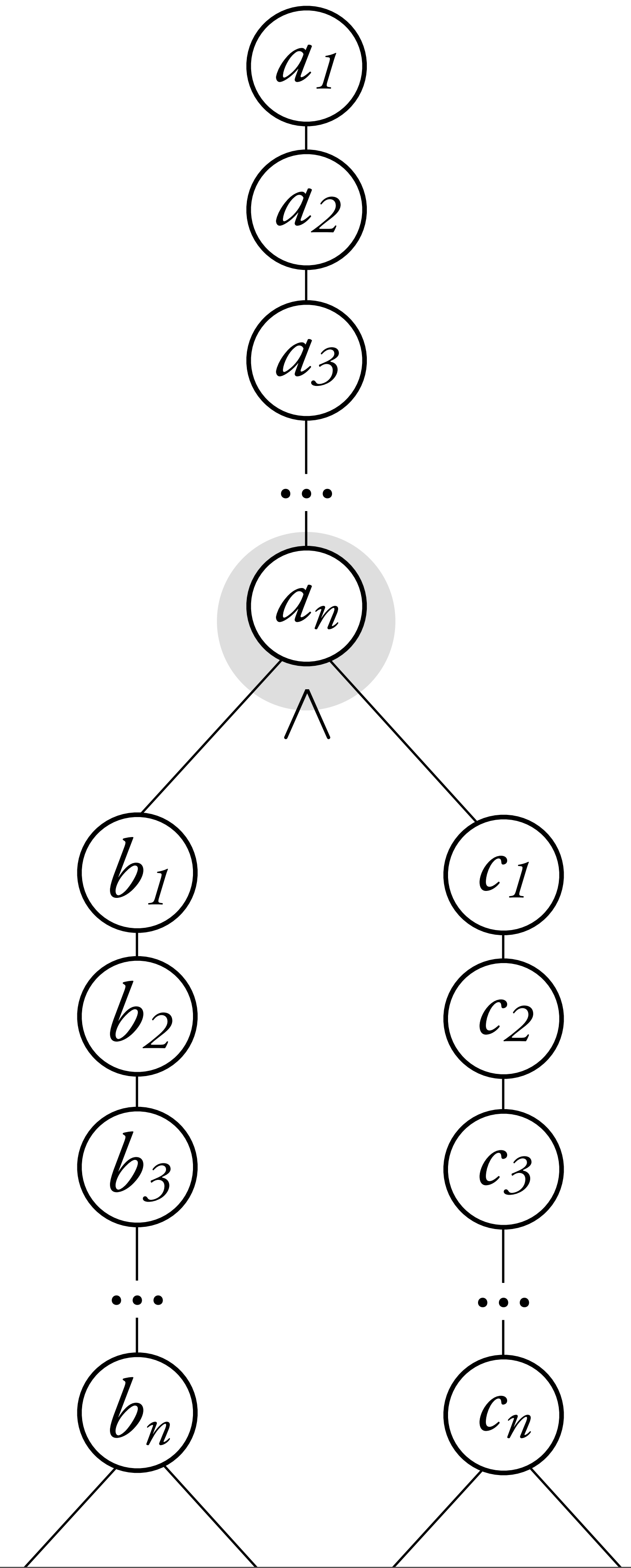


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

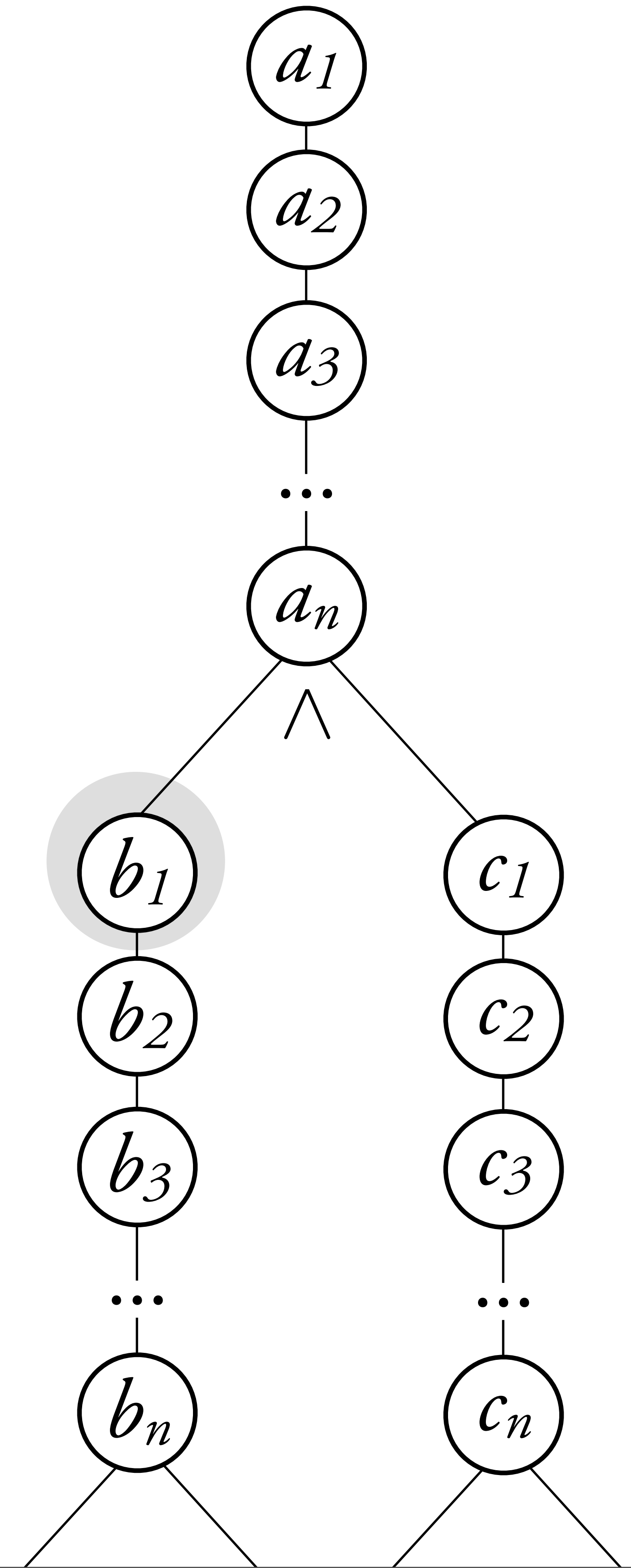


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

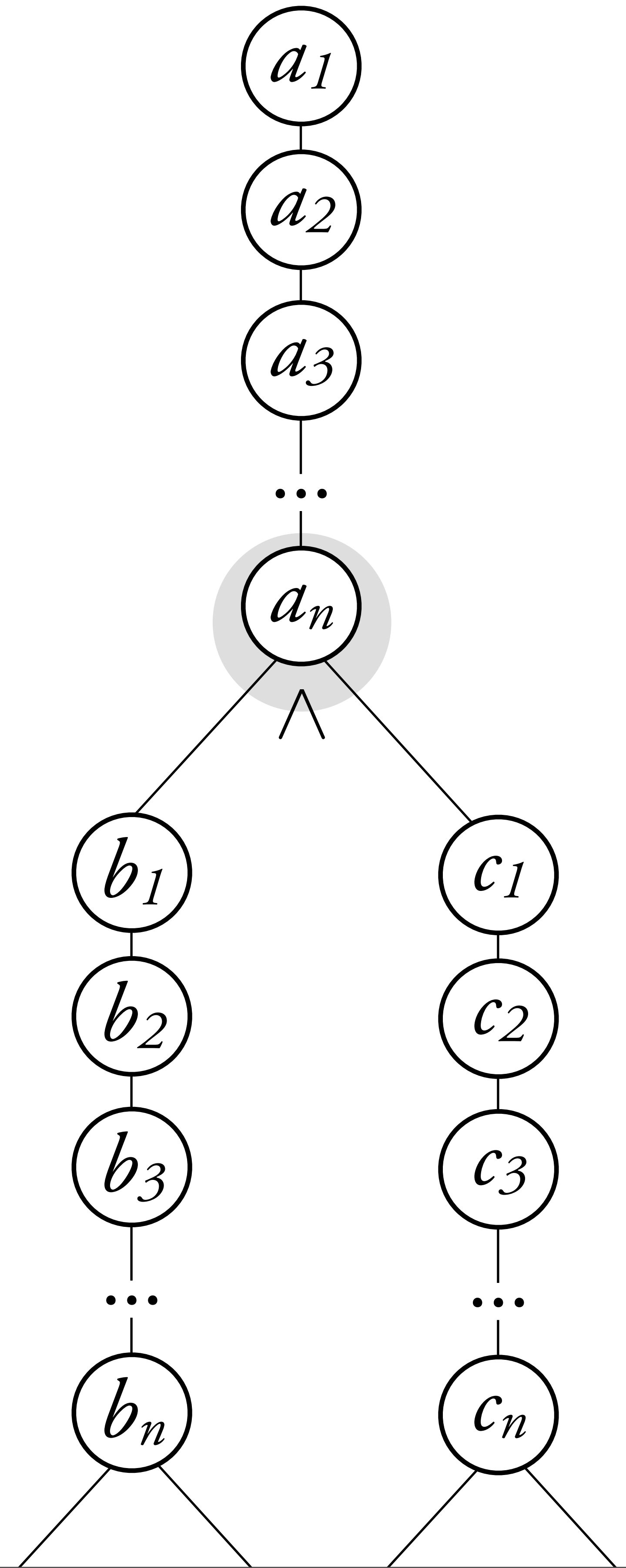


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

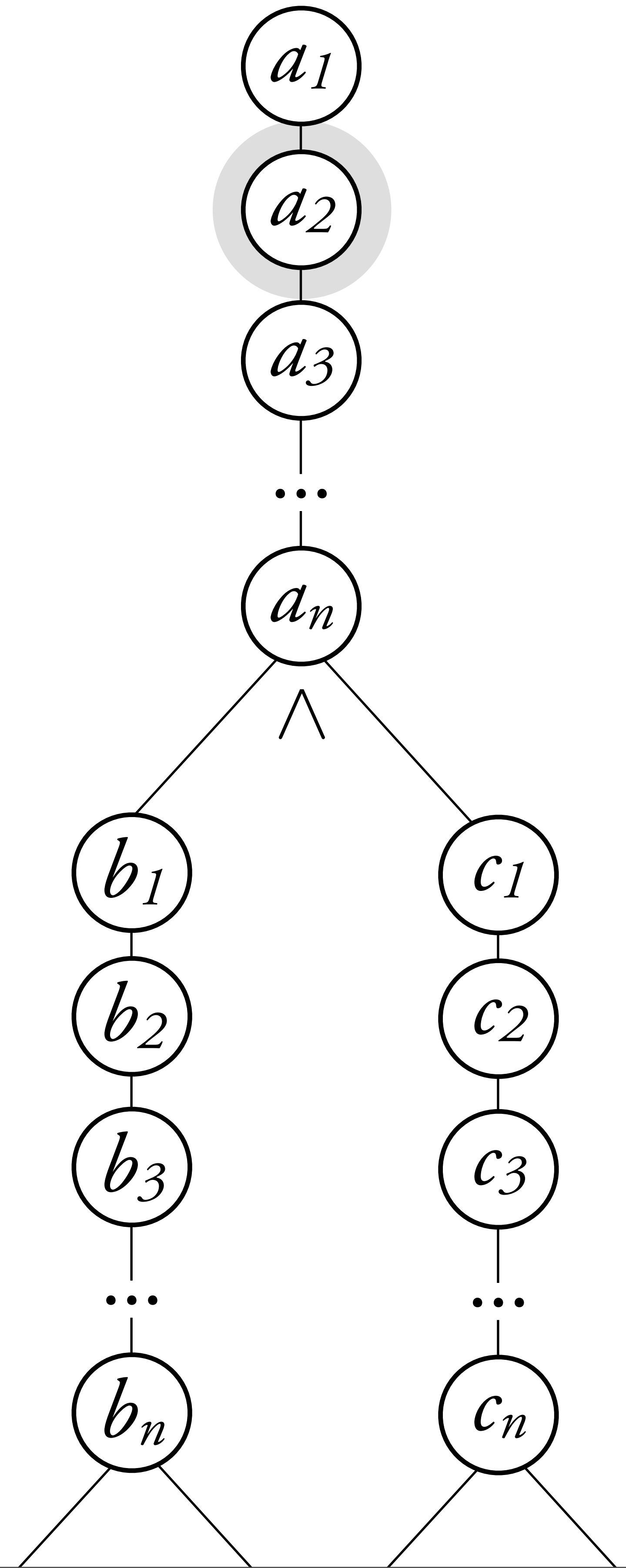


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

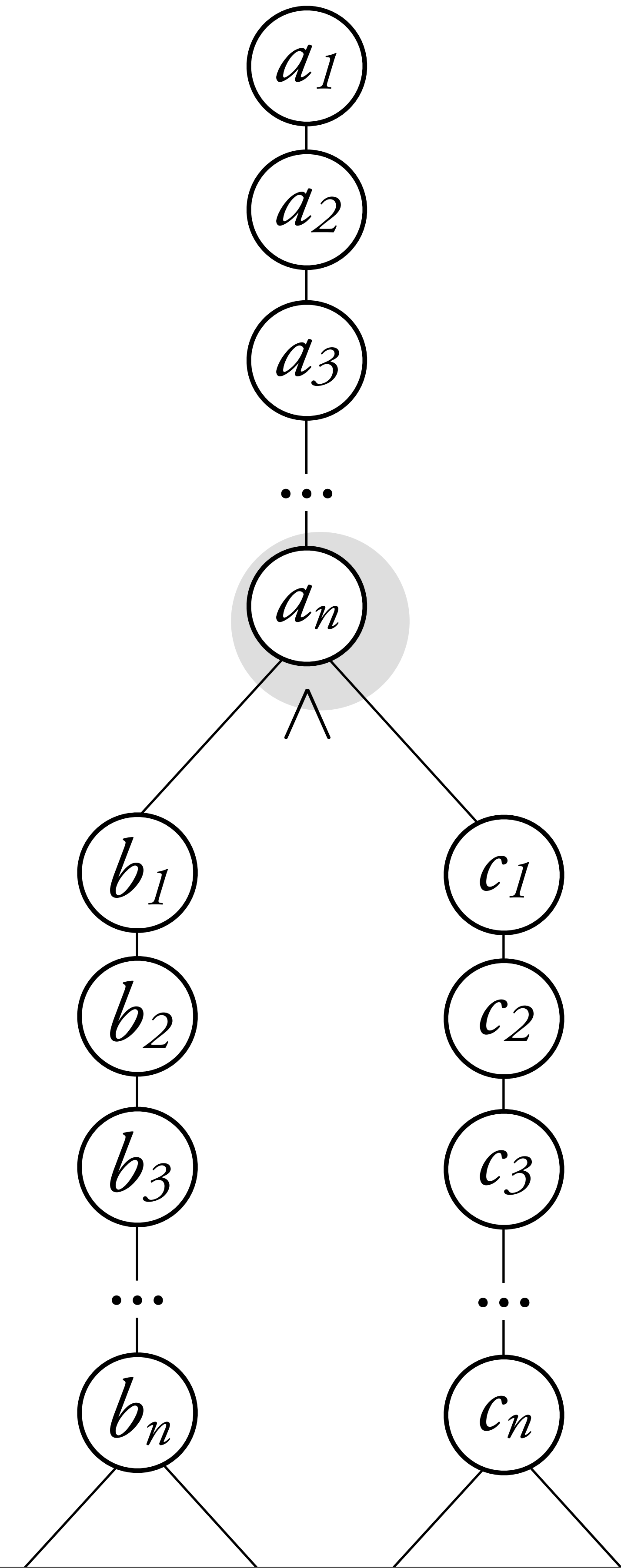


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses *n* memory cells, we write a tree-walking automaton with equivalent emptiness and *O(n)* states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

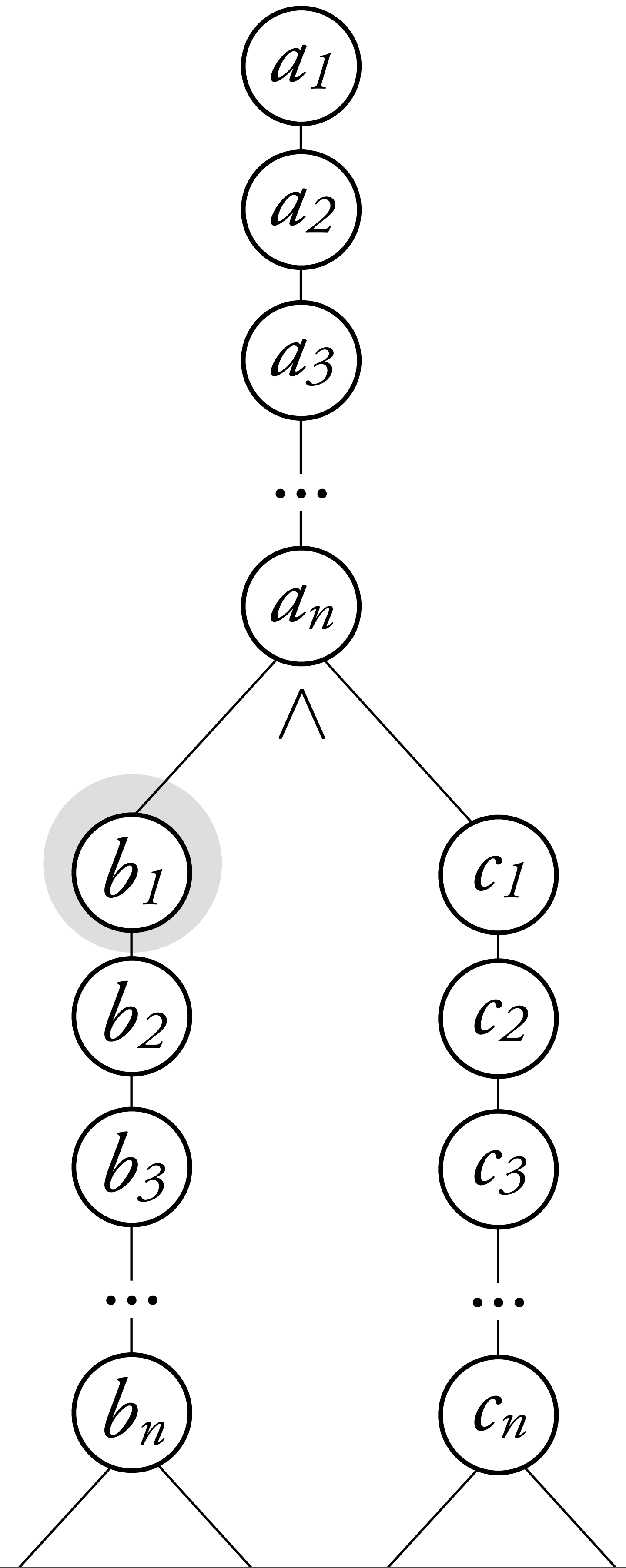


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

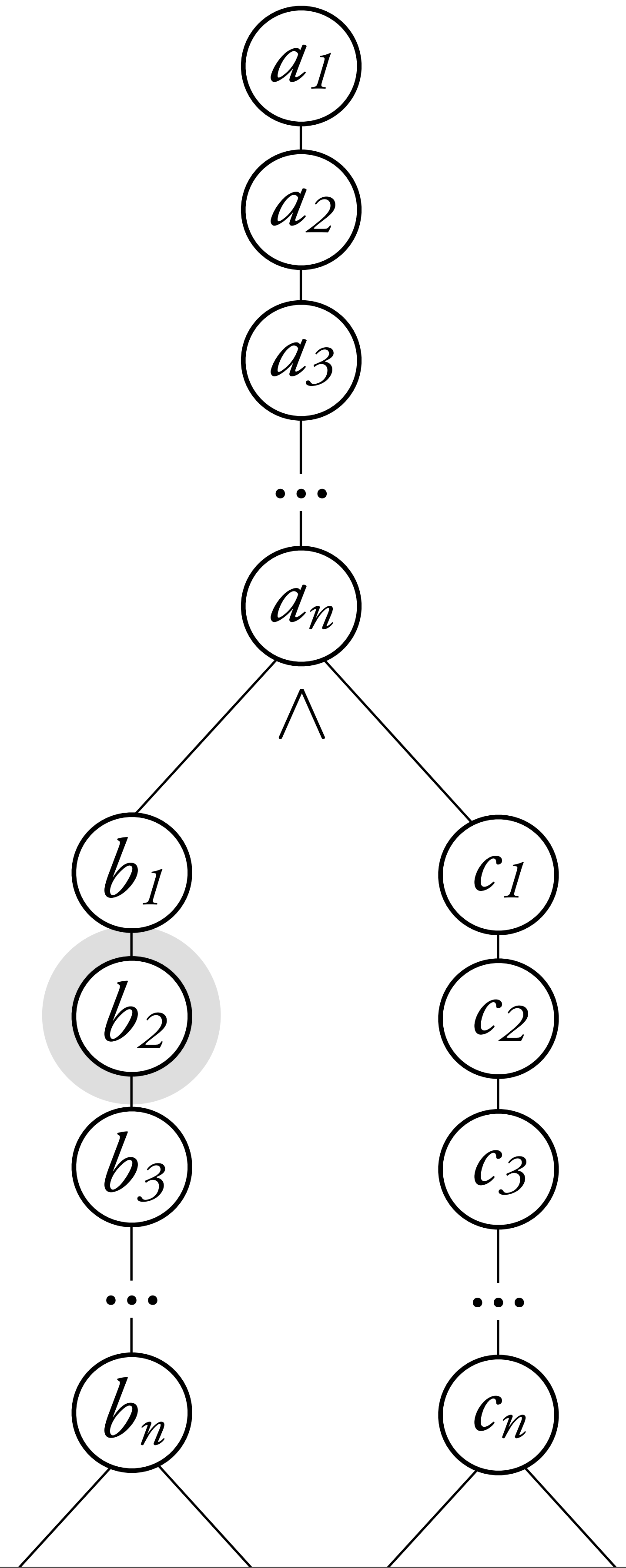


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

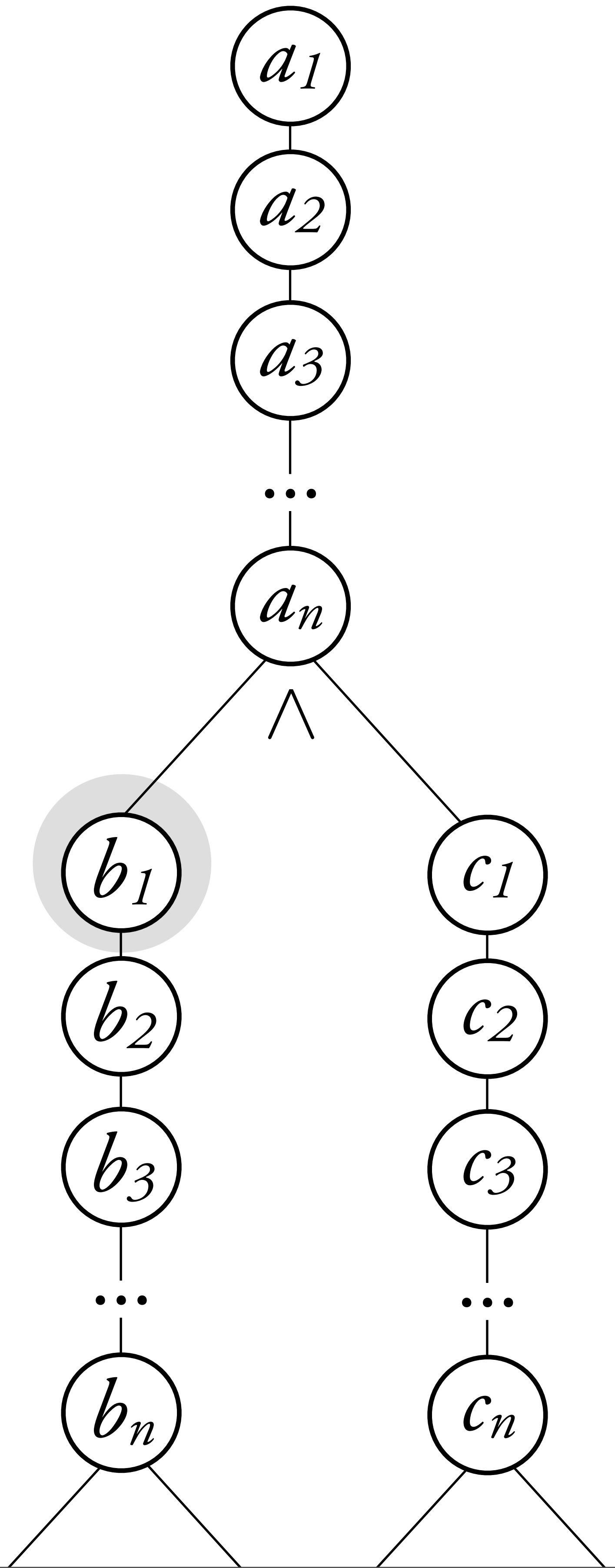


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

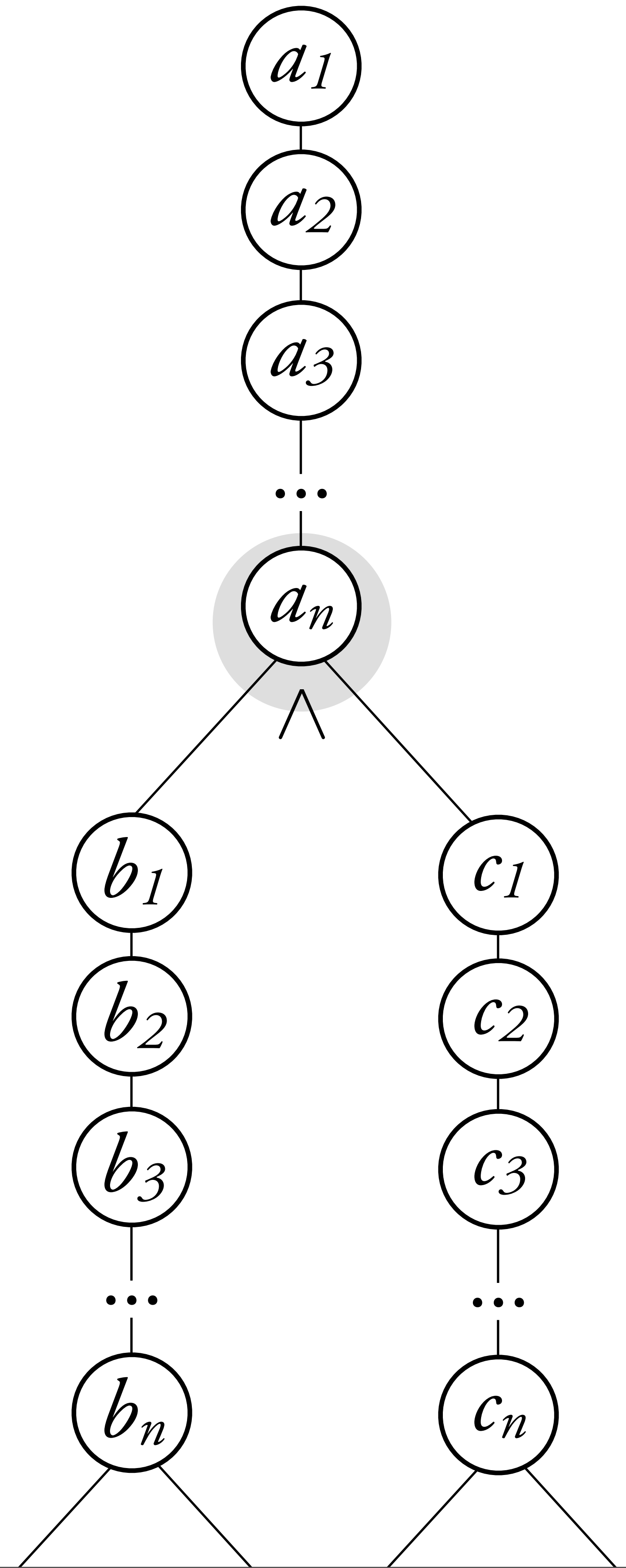
Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.



Theorem. Emptiness for tree-walking automata is EXPTIME-complete.



Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

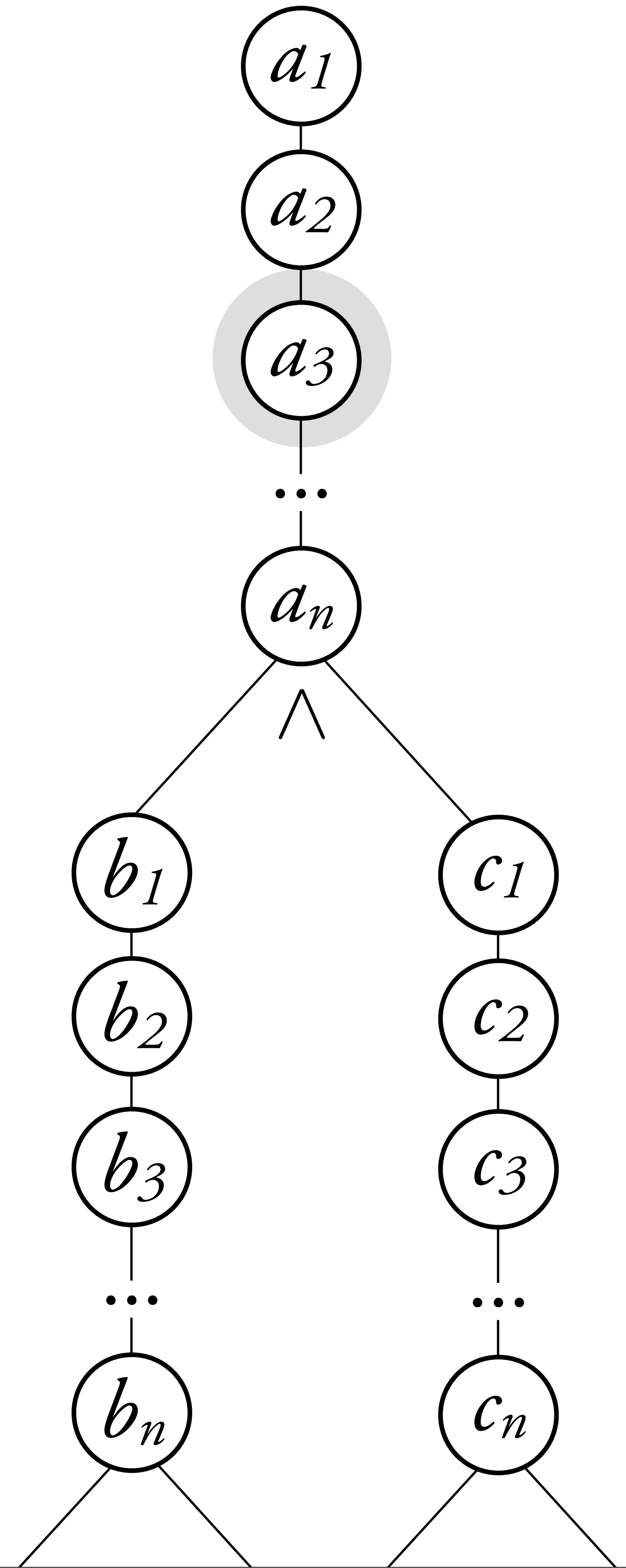
The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

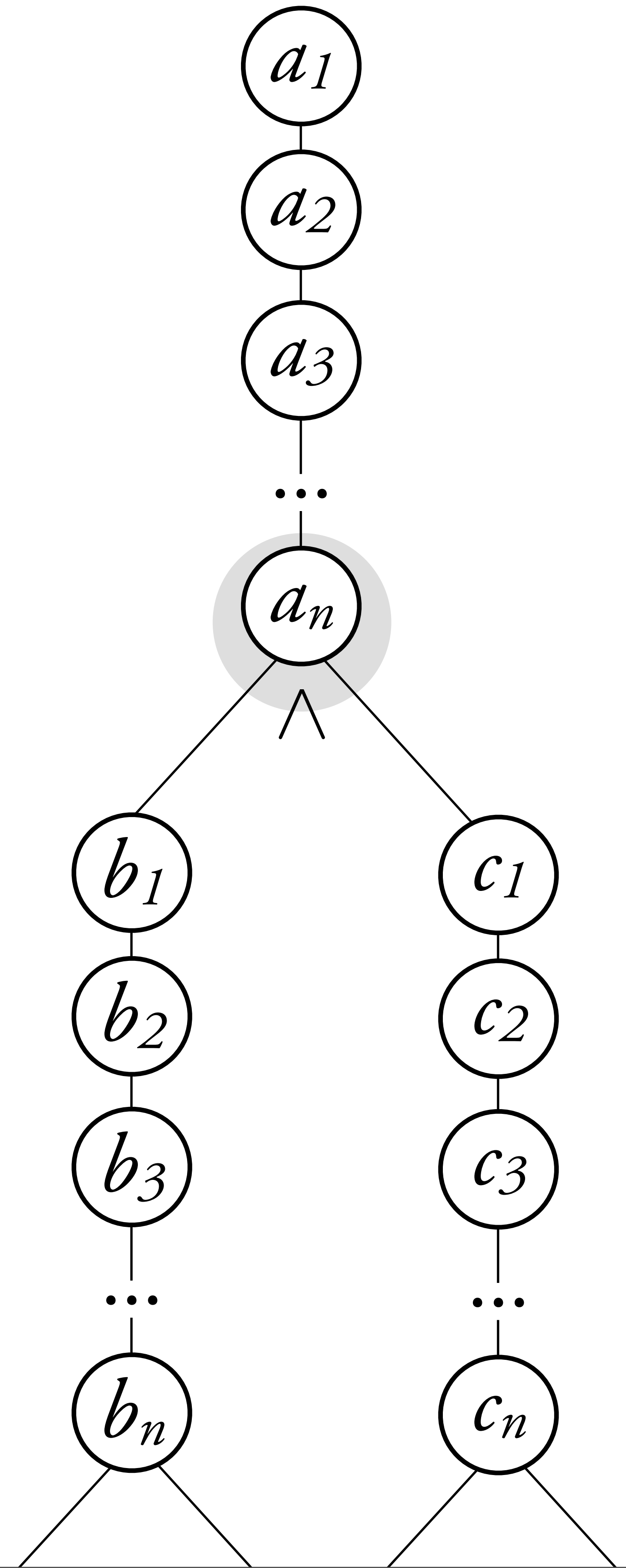


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

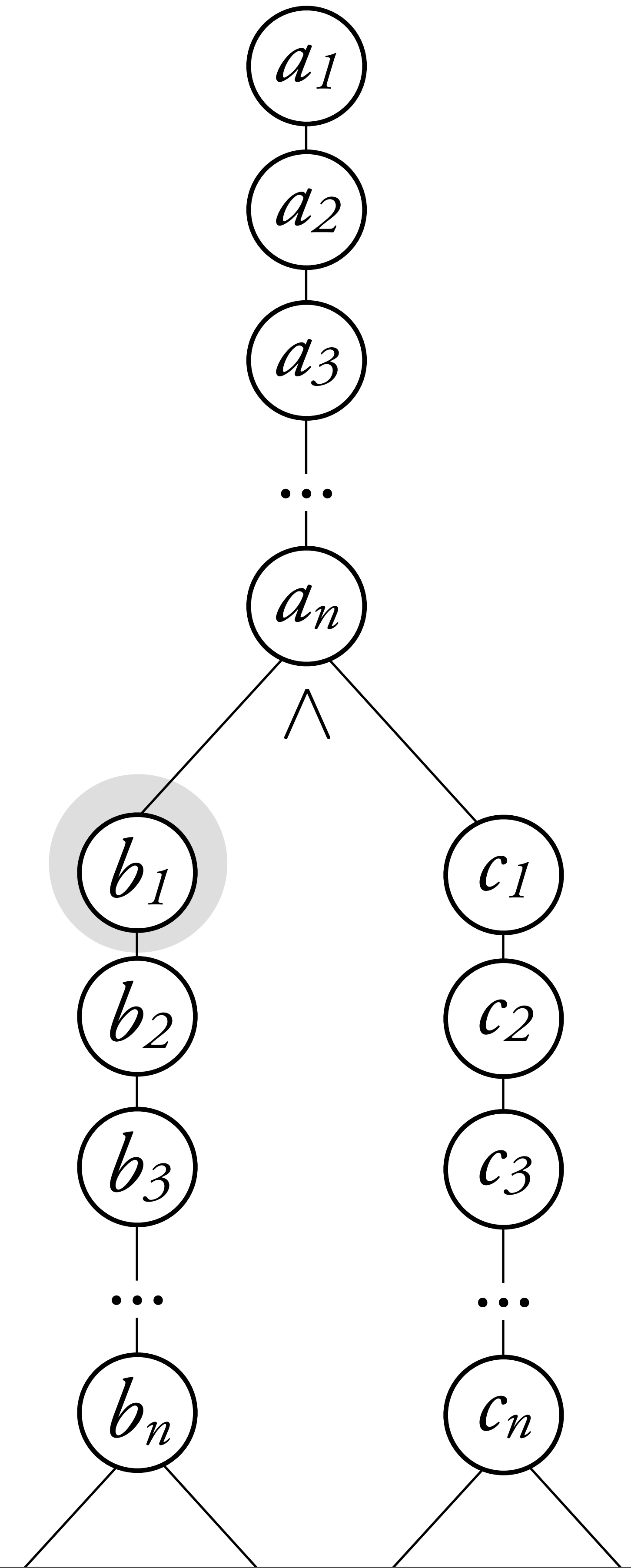


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.

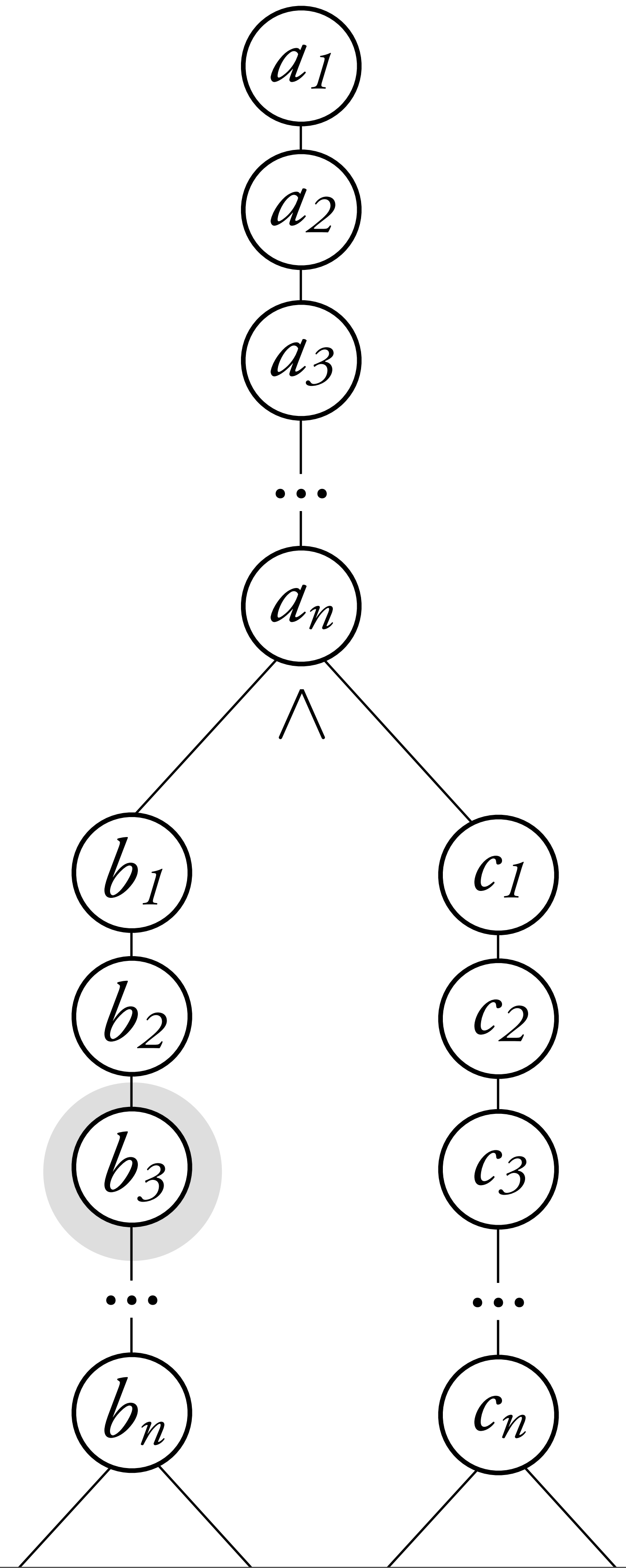


Theorem. Emptiness for tree-walking automata is EXPTIME-complete.

Hardness. Reduction from APSPACE.

For an alternating Turing machine that uses n memory cells, we write a tree-walking automaton with equivalent emptiness and $O(n)$ states.

The tree-walking automaton accepts computation trees. It does a DFS through the tree and checks that each branching is correct.



$\mathrm{TWA} \subseteq \mathrm{REG}$

Is the inclusion strict?

Theorem (B., Colcombet '05) The inclusion is strict.

Theorem (B., Colcombet '04)

Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$

$L \notin \mathrm{DTWA}$

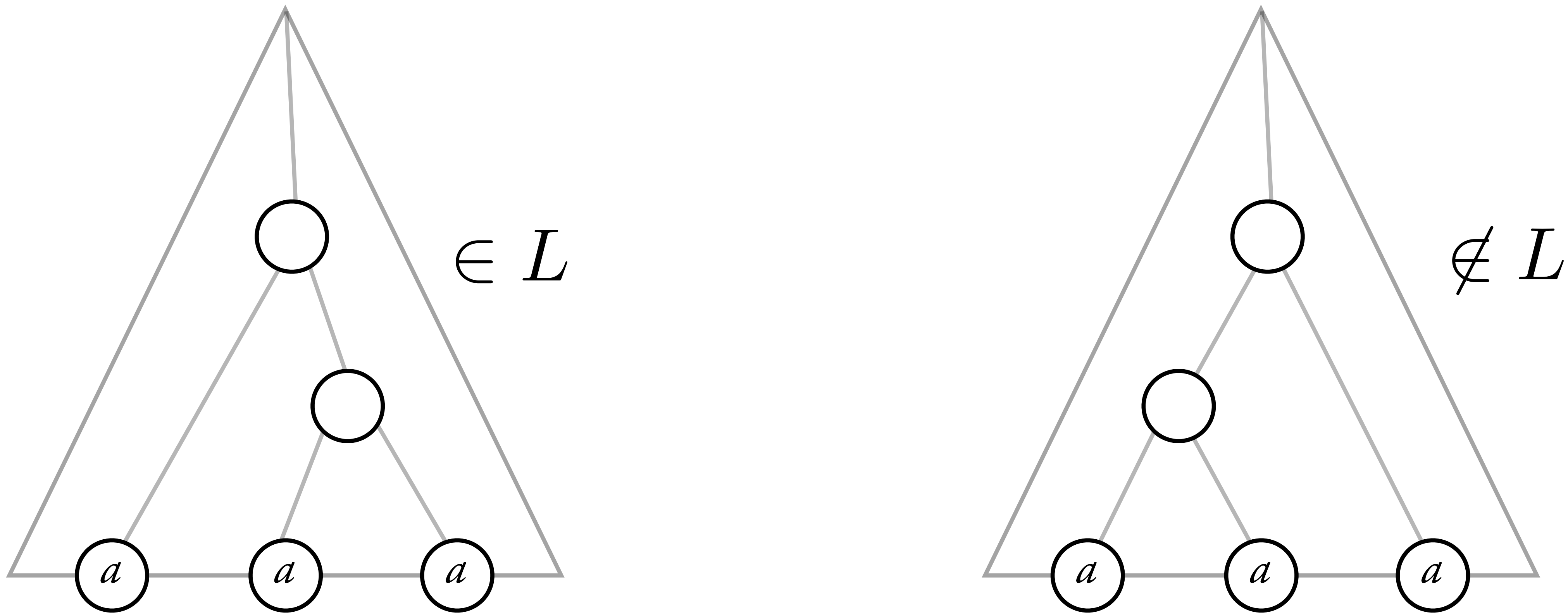
Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$

All nodes have label b, except three leaves with label a.

Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

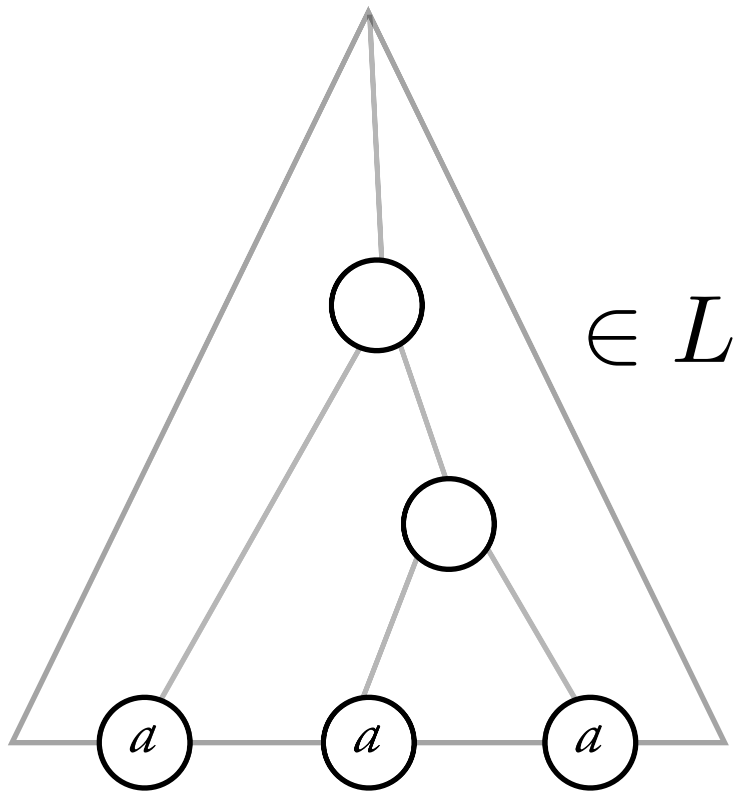
$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$



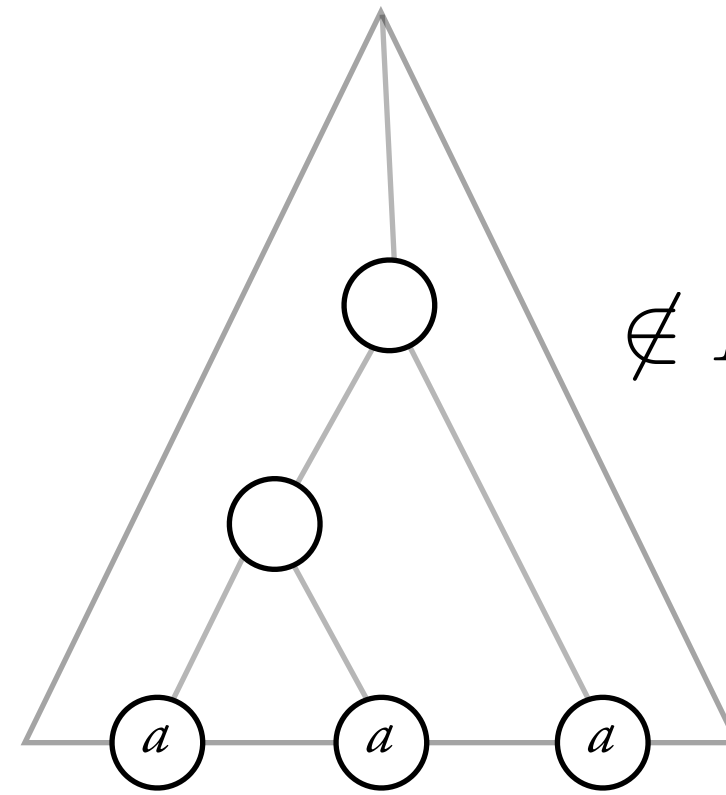
All nodes have label b, except three leaves with label a.

Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$



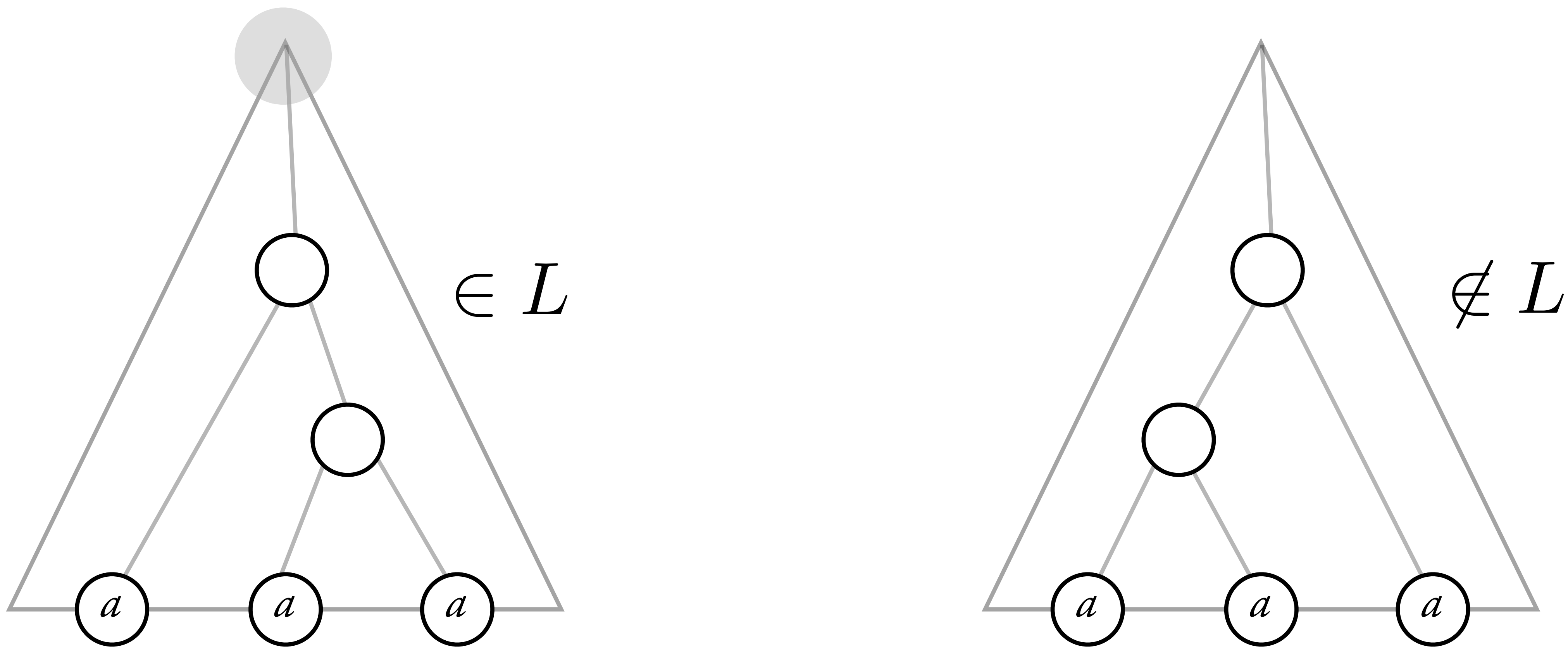
$\in L$



$\notin L$

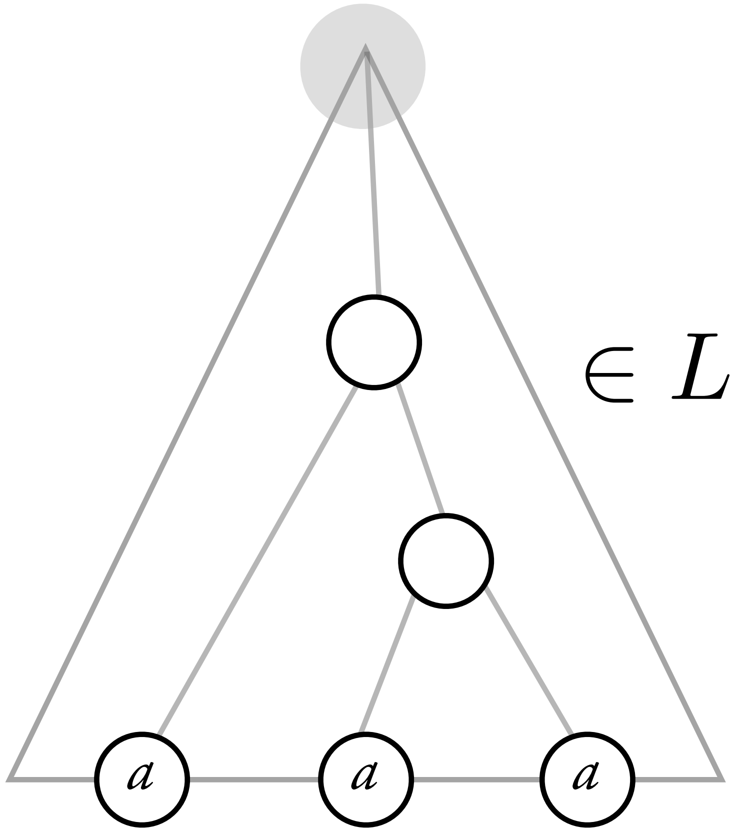
Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

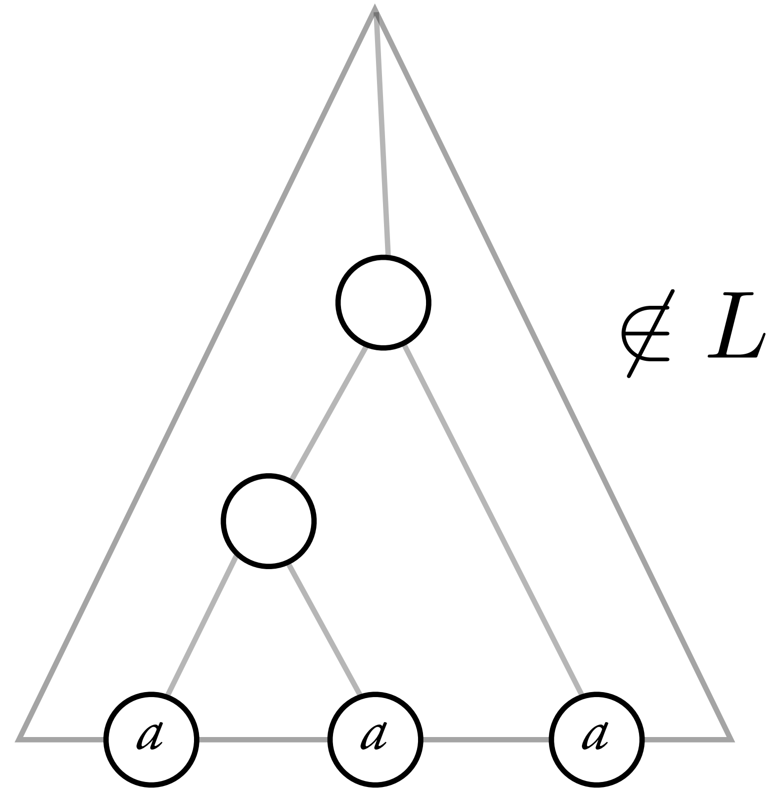
$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$



Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$

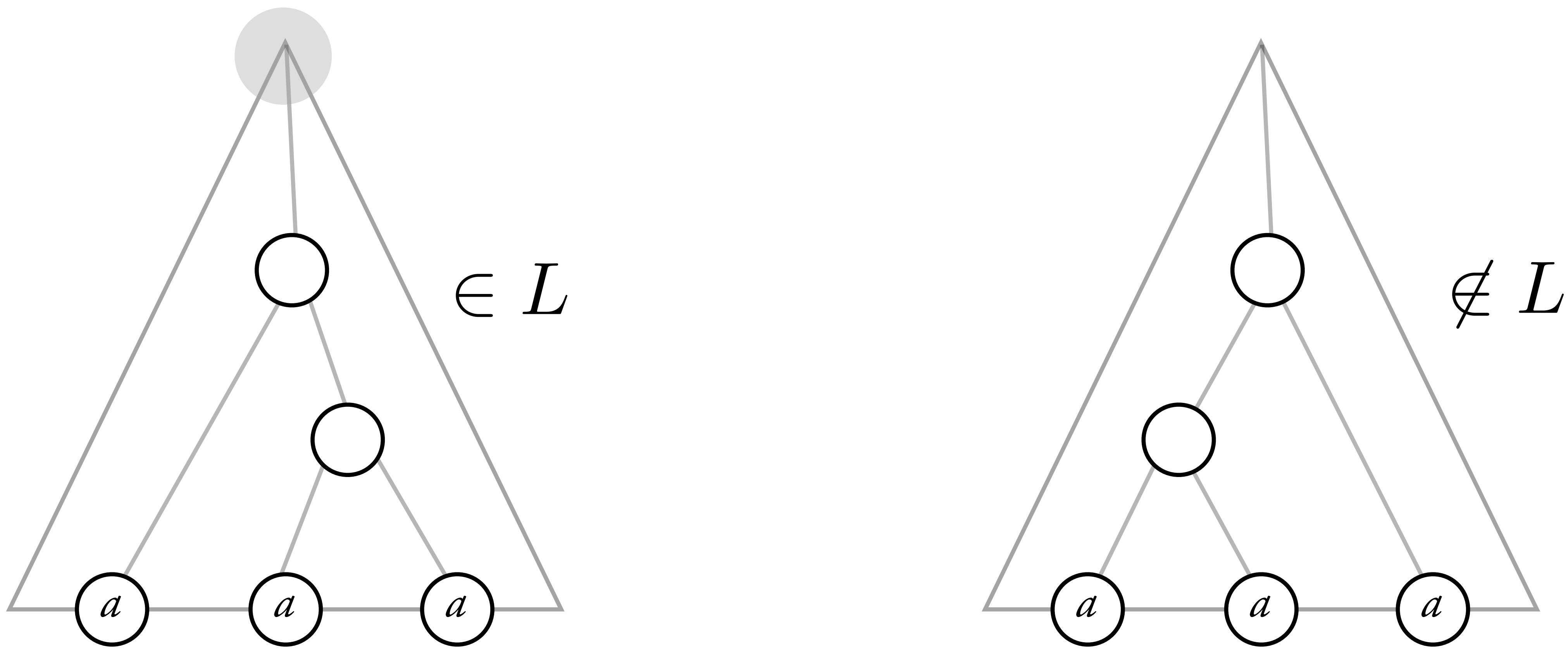




Using DFS, check that all nodes have b, except three leaves with a.

Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$



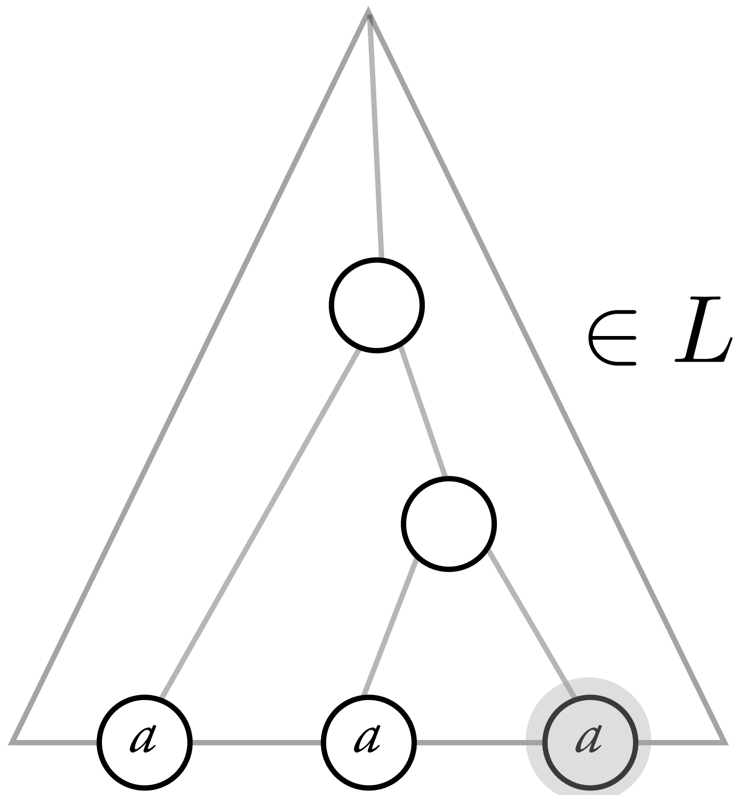
Using DFS, check that all nodes have *b*, except three leaves with *a*.
Go to the rightmost *a*.

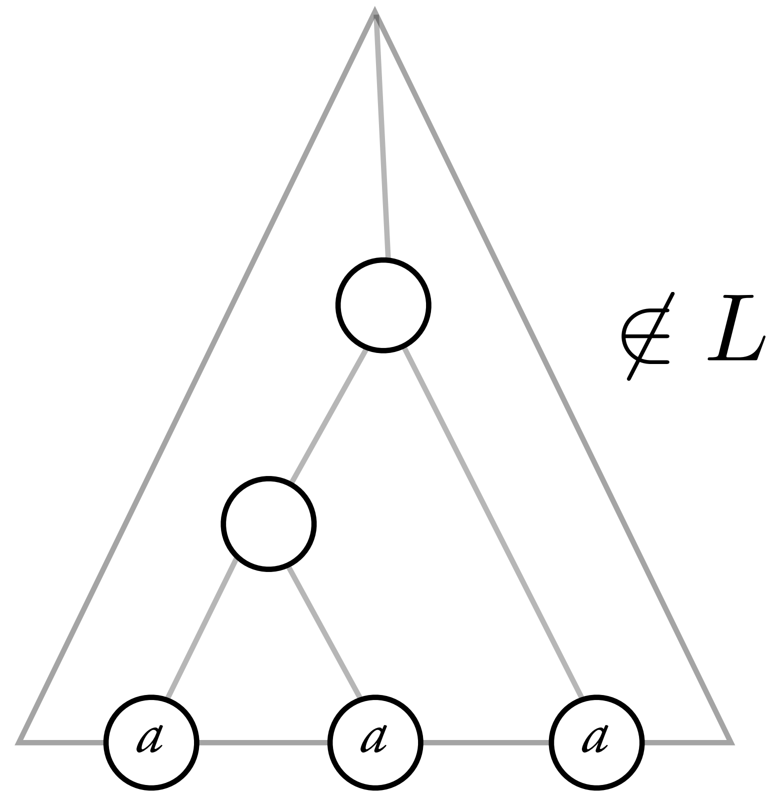
Theorem (B., Colcombet '04)

Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$

$L \notin \mathrm{DTWA}$



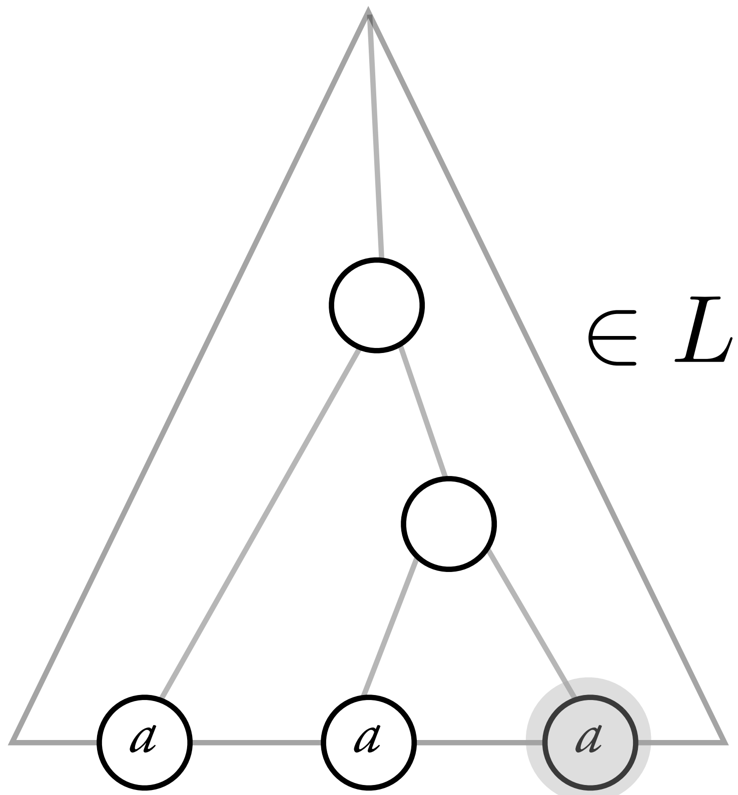


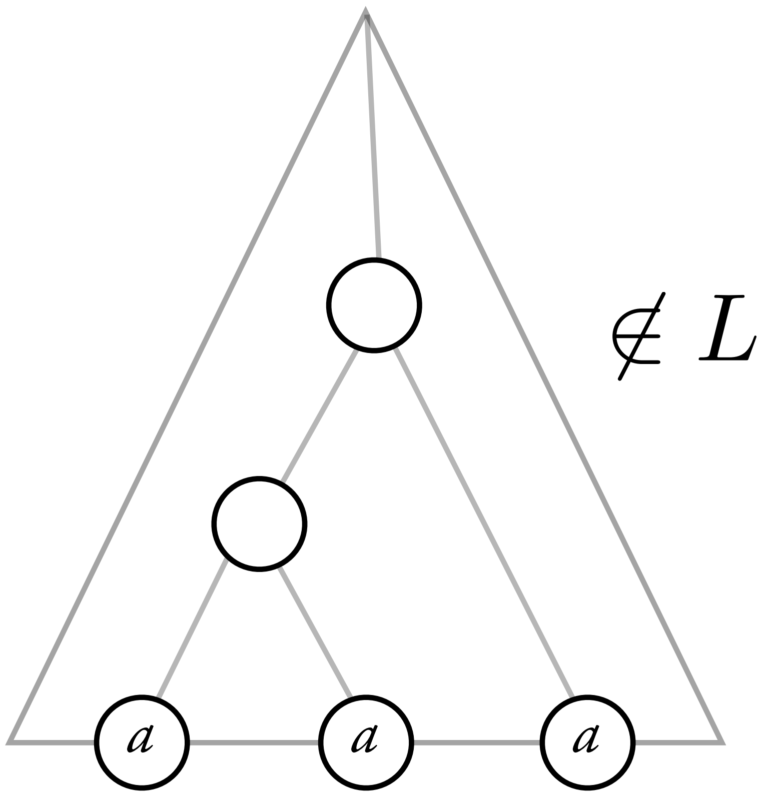
Using DFS, check that all nodes have *b*, except three leaves with *a*.

Go to the rightmost *a*.

Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$

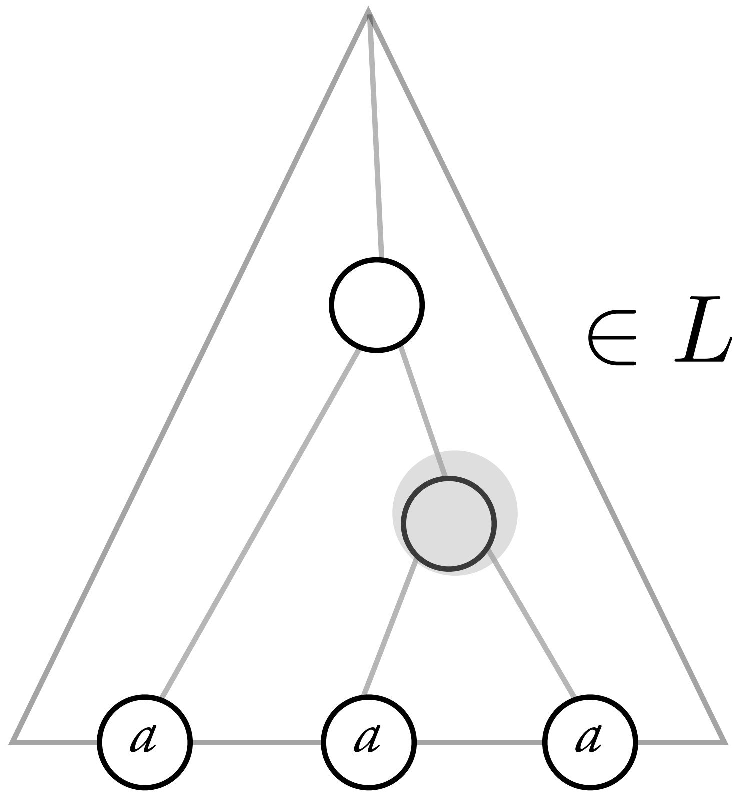


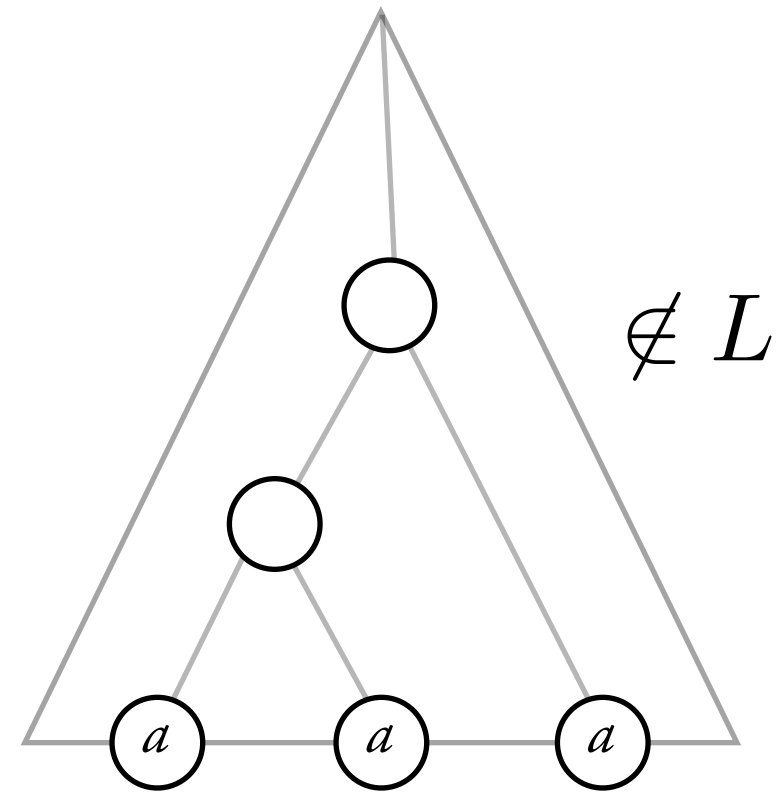


Using DFS, check that all nodes have b, except three leaves with a.
Go to the rightmost a.
Nondeterministically pick an ancestor.

Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$





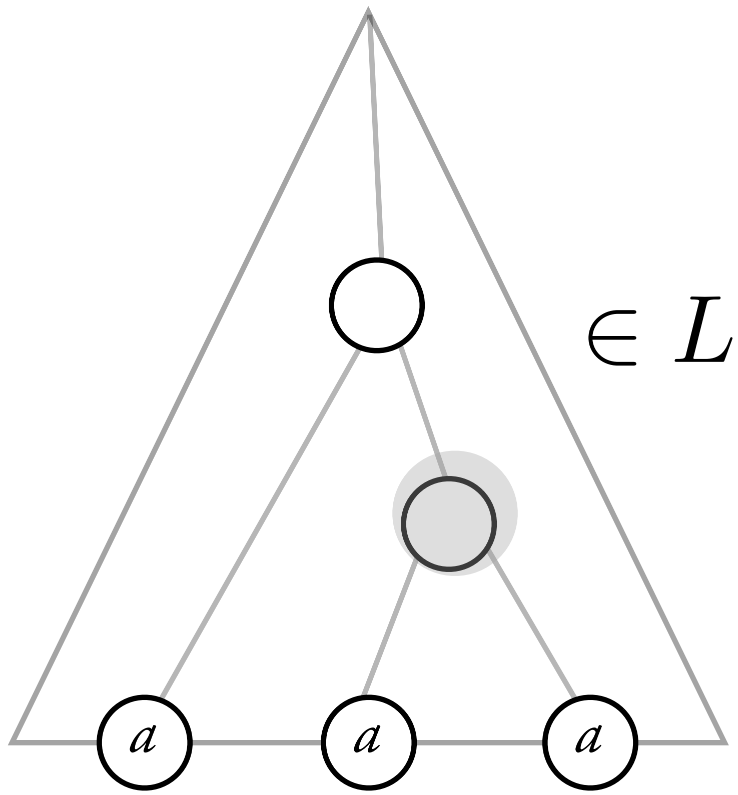
Using DFS, check that all nodes have b, except three leaves with a.

Go to the rightmost a.

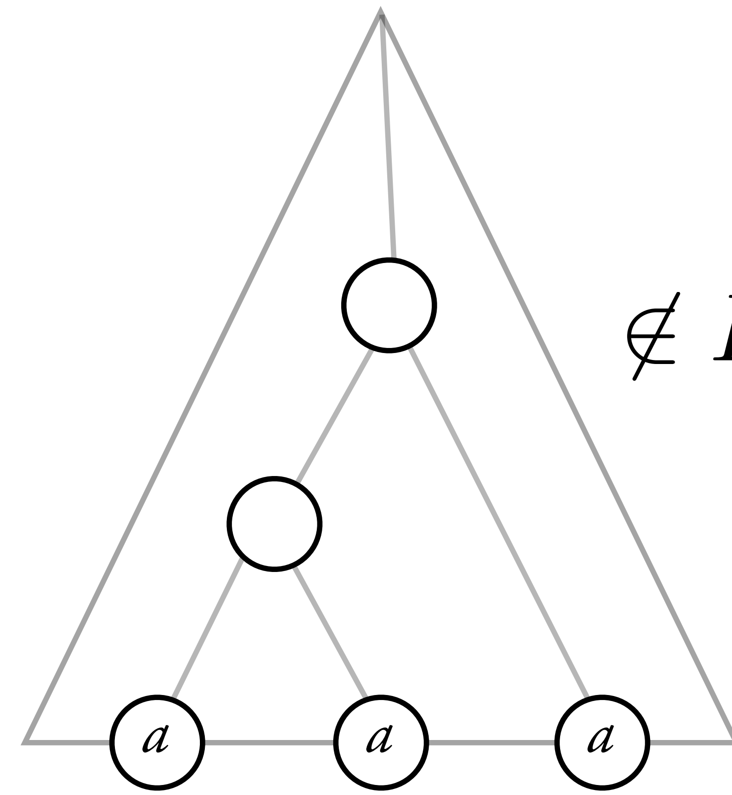
Nondeterministically pick an ancestor.

Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$



$\in L$

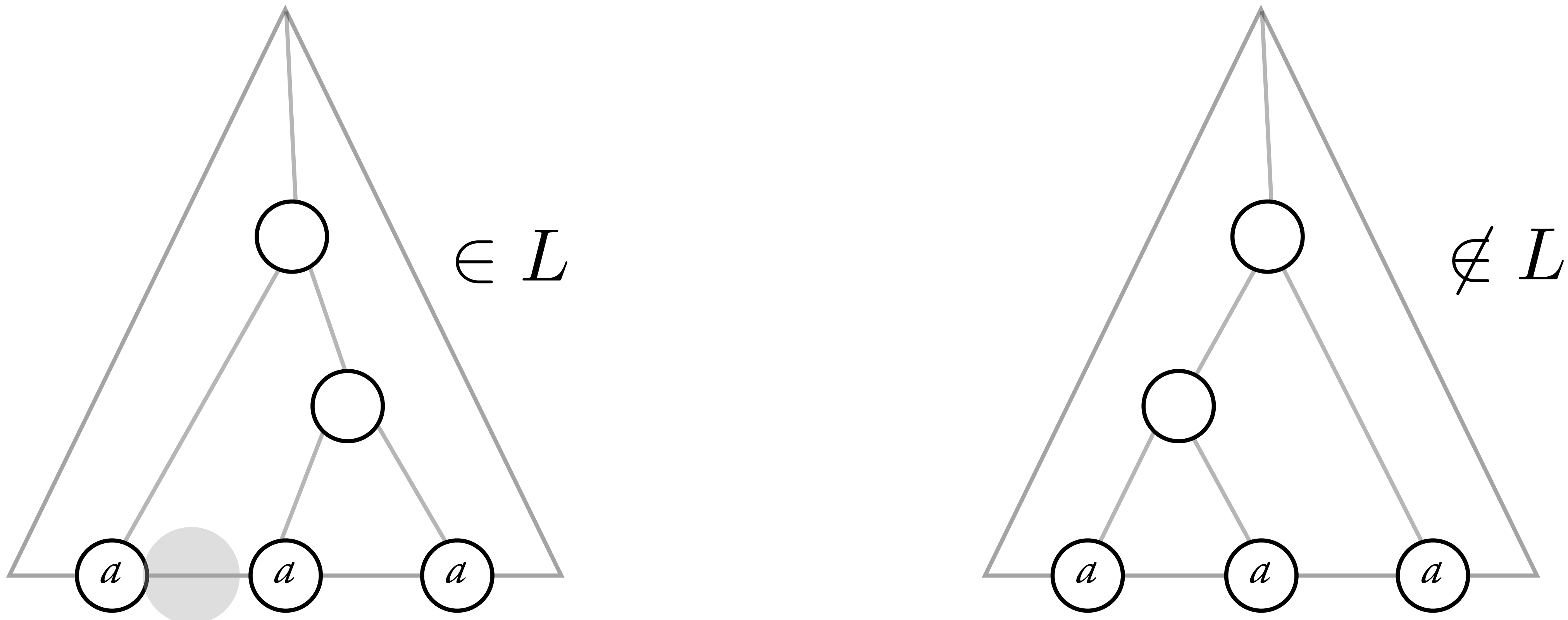


$\notin L$

Using DFS, check that all nodes have *b*, except three leaves with *a*.
Go to the rightmost *a*.
Nondeterministically pick an ancestor.
Descend to the leaf on the leftmost path.

Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$



Using DFS, check that all nodes have *b*, except three leaves with *a*.

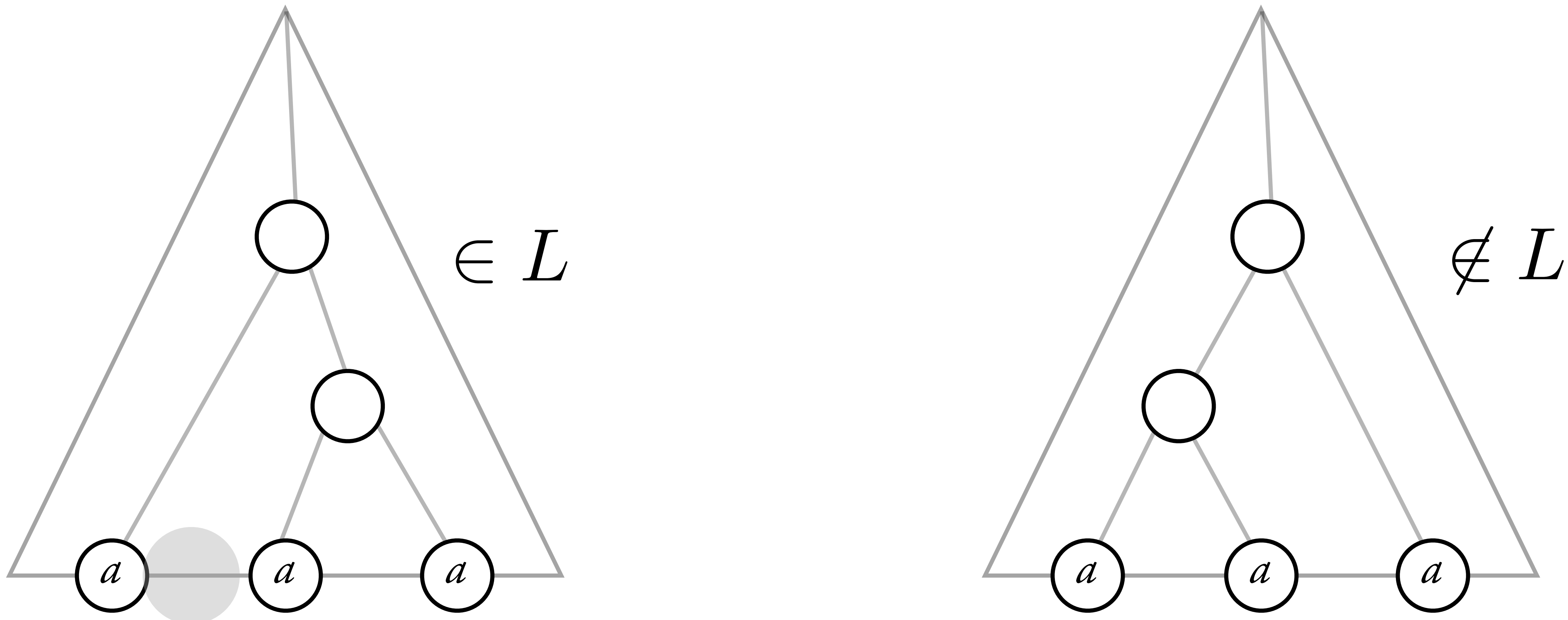
Go to the rightmost *a*.

Nondeterministically pick an ancestor.

Descend to the leaf on the leftmost path.

Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$



Using DFS, check that all nodes have *b*, except three leaves with *a*.

Go to the rightmost *a*.

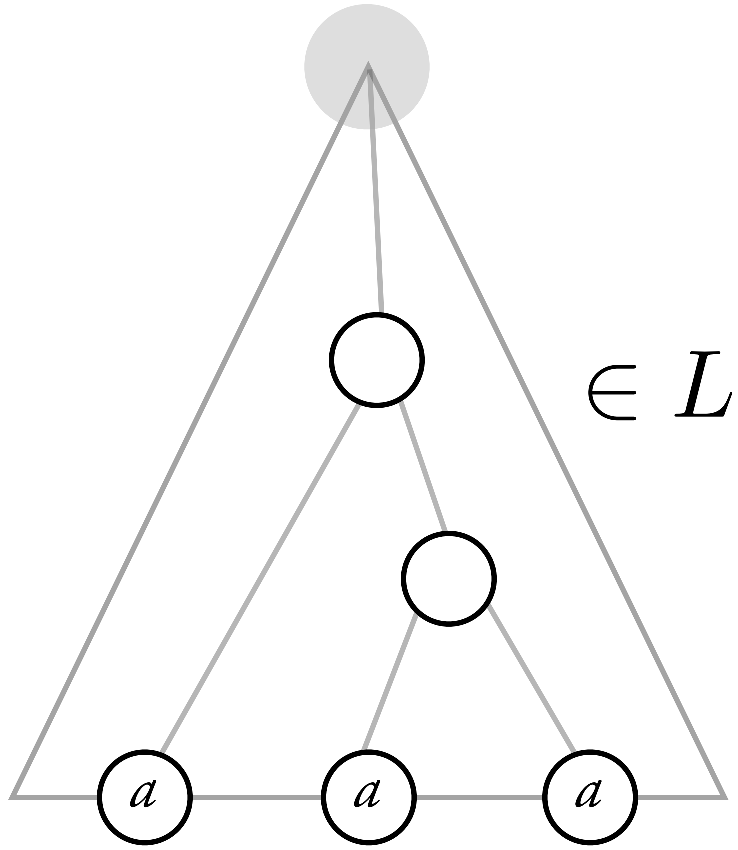
Nondeterministically pick an ancestor.

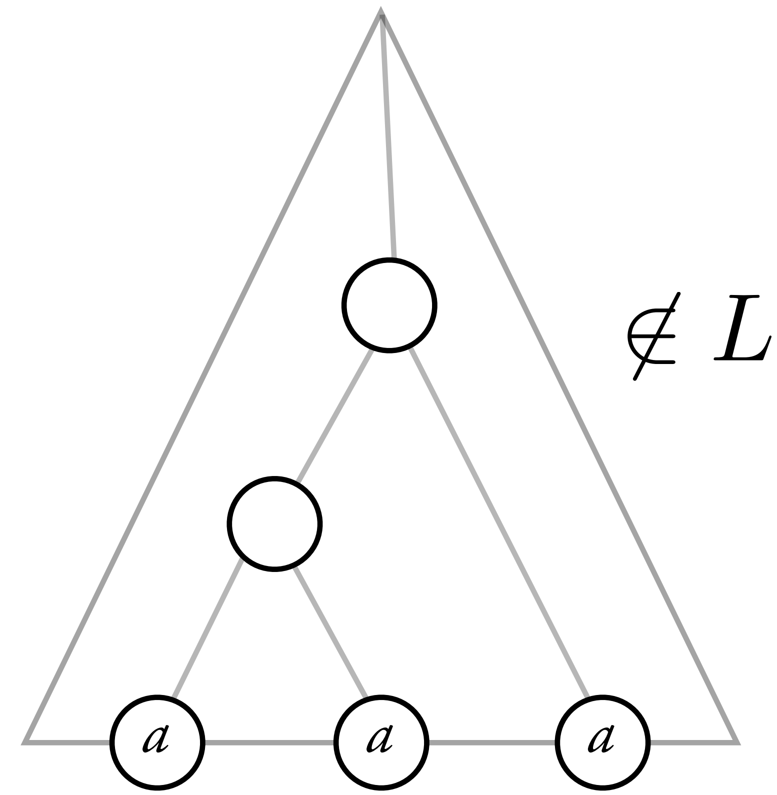
Descend to the leaf on the leftmost path.

Accept if there are exactly two *a*'s to the right.

Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$
$L \notin \mathrm{DTWA}$





Using DFS, check that all nodes have *b*, except three leaves with *a*.

Go to the rightmost *a*.

Nondeterministically pick an ancestor.

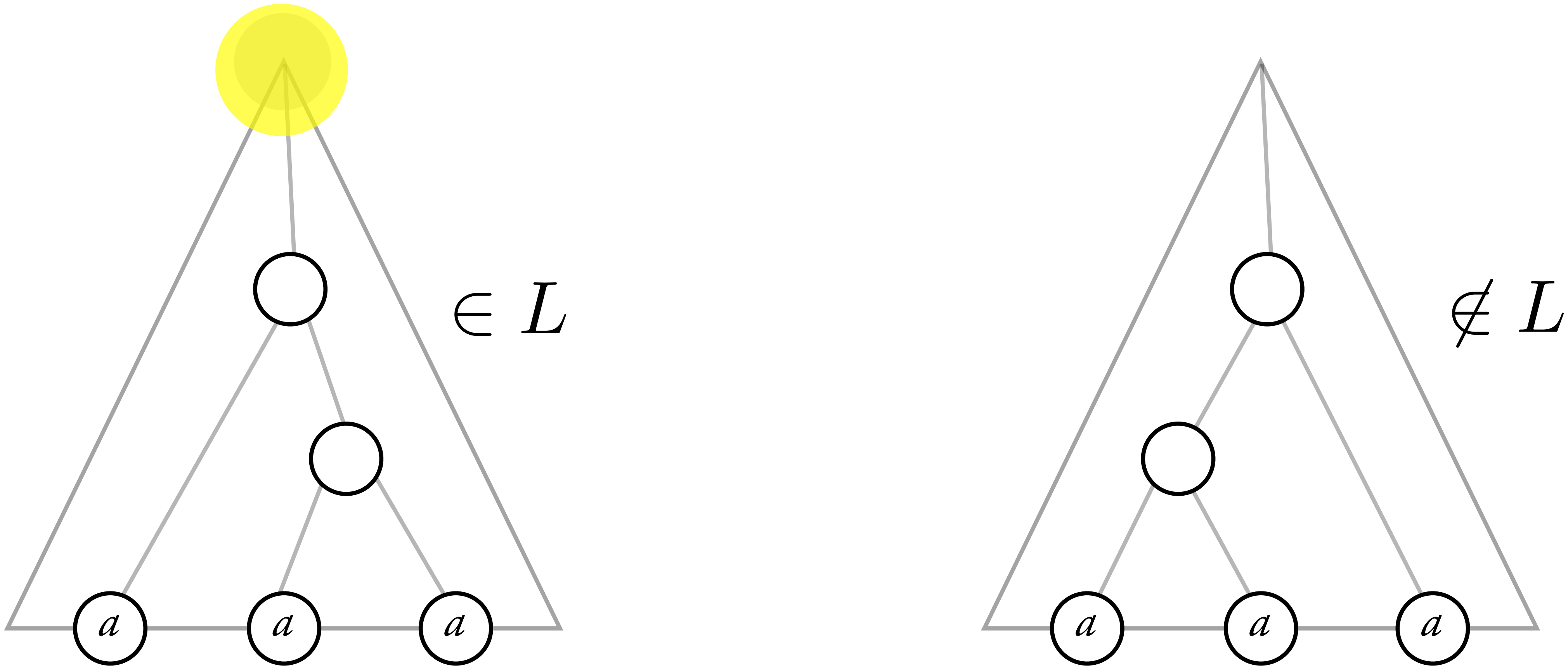
Descend to the leaf on the leftmost path.

Accept if there are exactly two *a*'s to the right.

Theorem (B., Colcombet '04)
Tree-walking automata cannot be determinized.

$L \in \mathrm{TWA}$

$L \notin \mathrm{DTWA}$



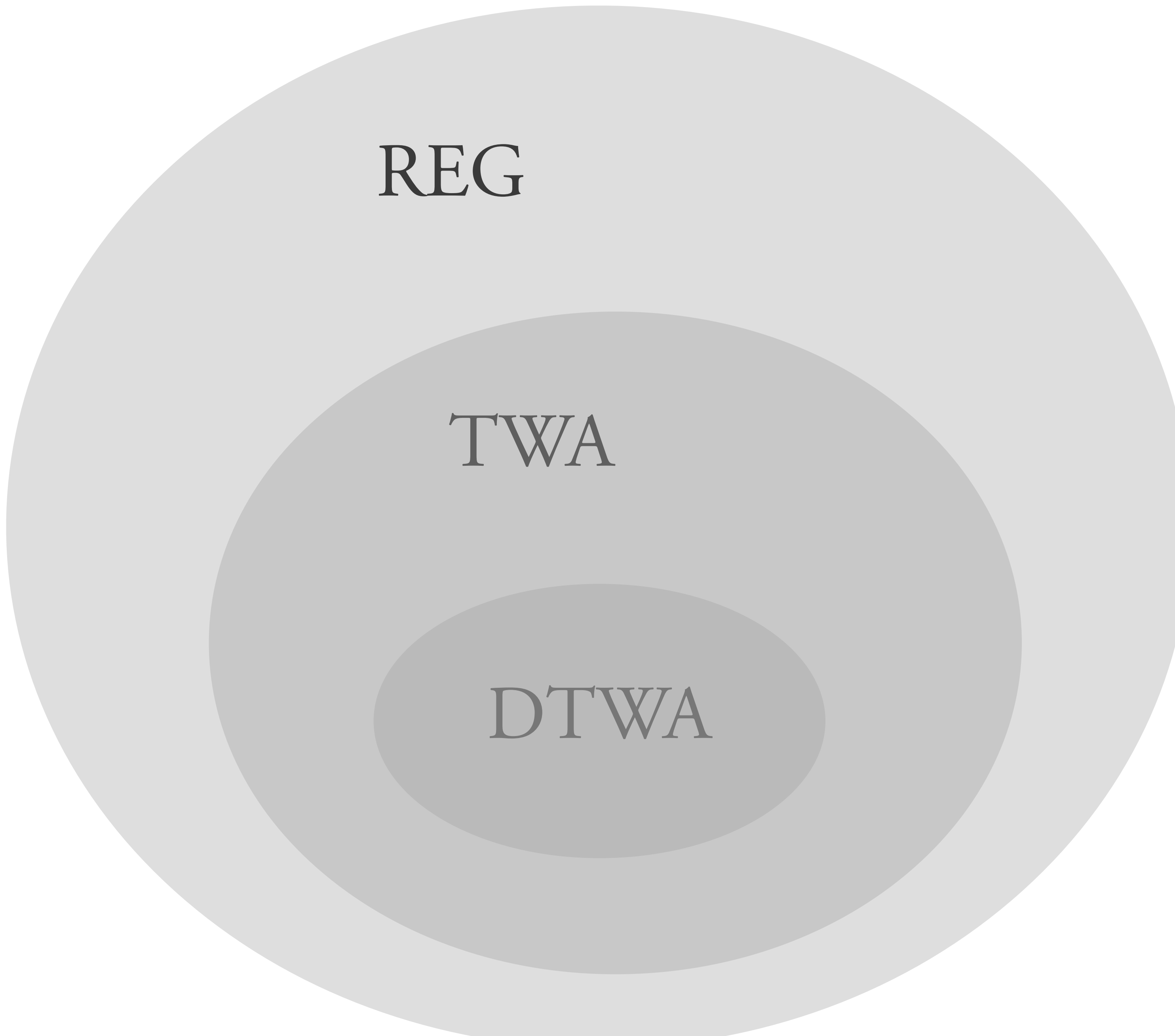
Using DFS, check that all nodes have b, except three leaves with a.

Go to the rightmost a.

Nondeterministically pick an ancestor.

Descend to the leaf on the leftmost path.

Accept if there are exactly two a's to the right.


REG
TWA
DTWA

Plan

-A tree-walking automaton

-Expressive power

-Pebble automata and logic

Plan

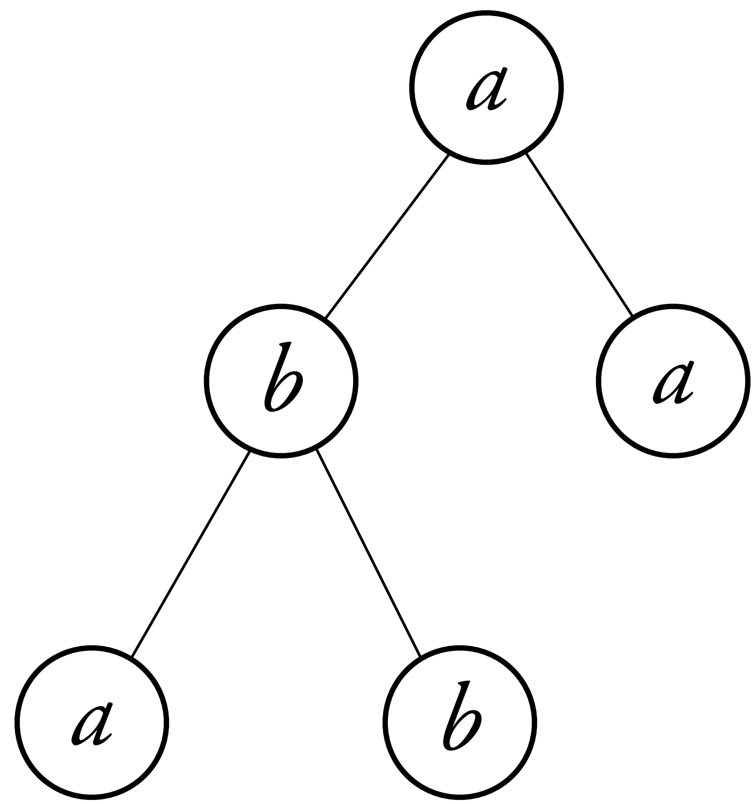
-A tree-walking automaton

-Expressive power

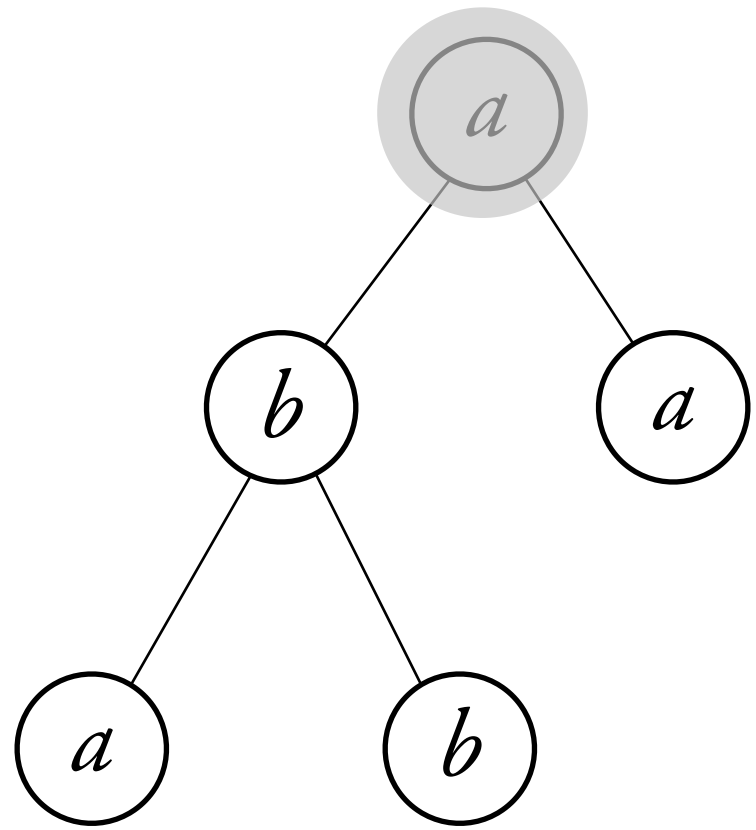
-Pebble automata and logic

In a large tree with only one type of label, all nodes look the same.
What if the automaton could mark nodes with pebbles?

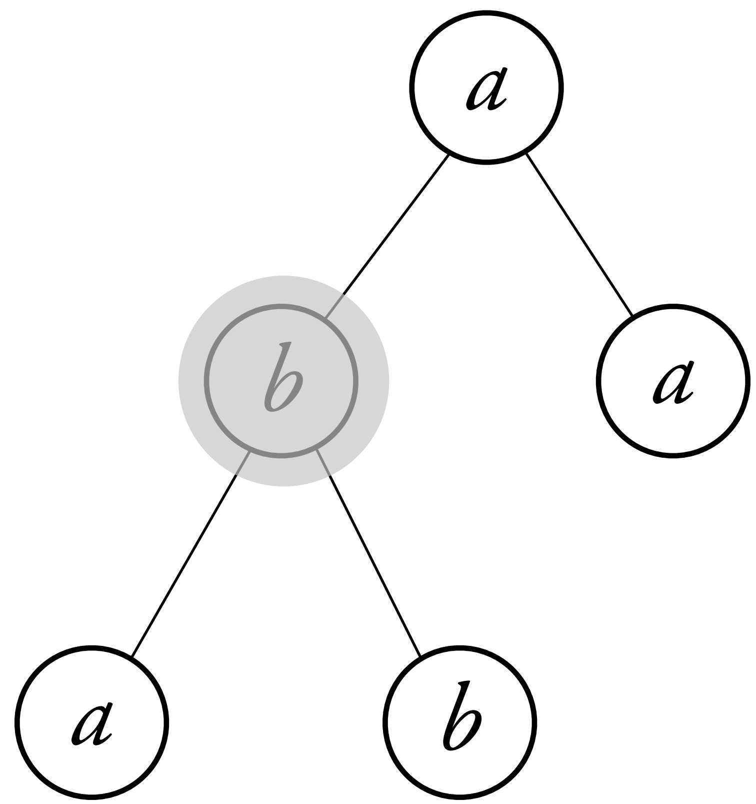
In a large tree with only one type of label, all nodes look the same.
What if the automaton could mark nodes with pebbles?



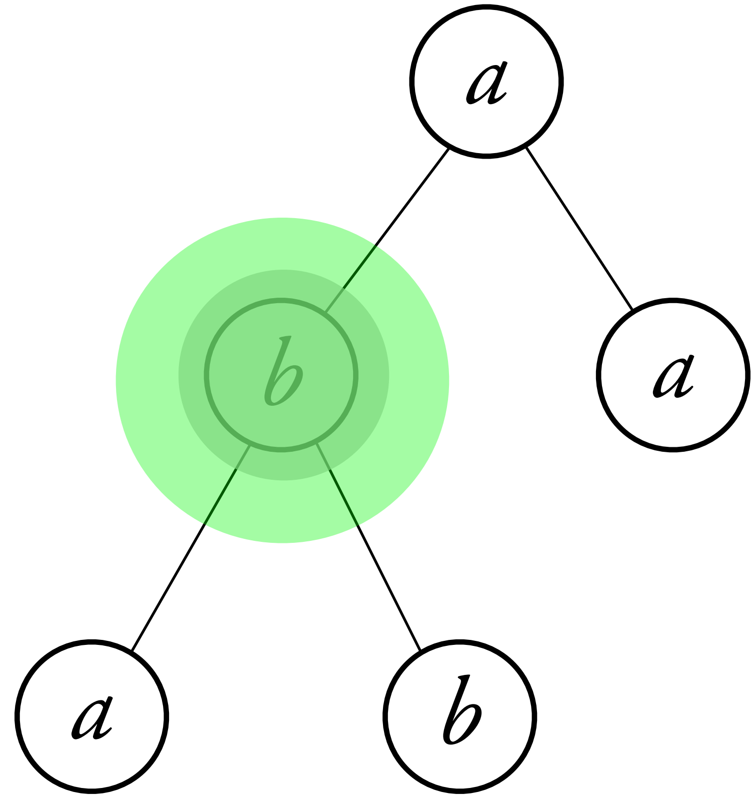
In a large tree with only one type of label, all nodes look the same.
What if the automaton could mark nodes with pebbles?



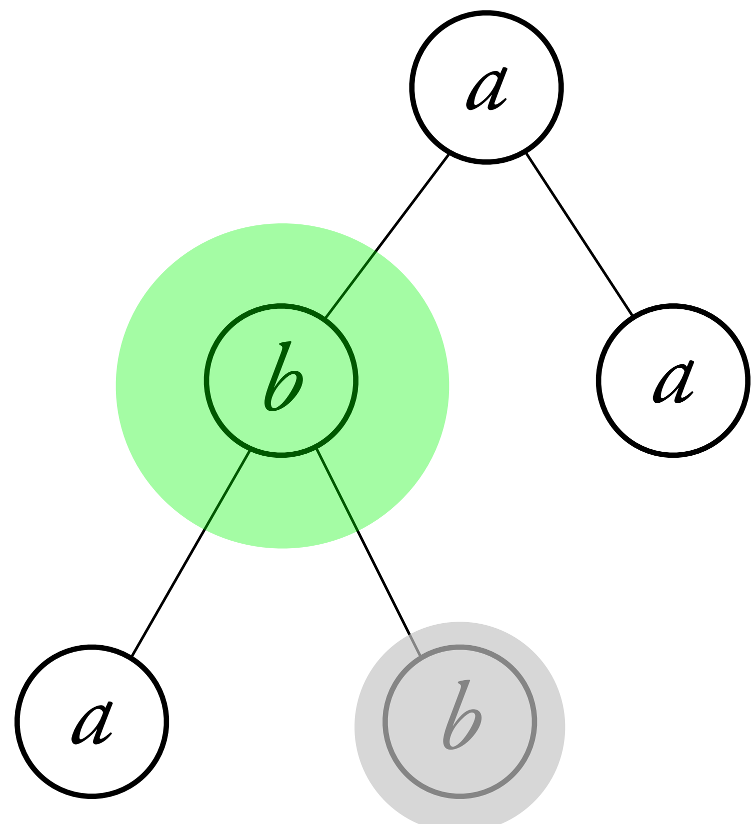
In a large tree with only one type of label, all nodes look the same.
What if the automaton could mark nodes with pebbles?



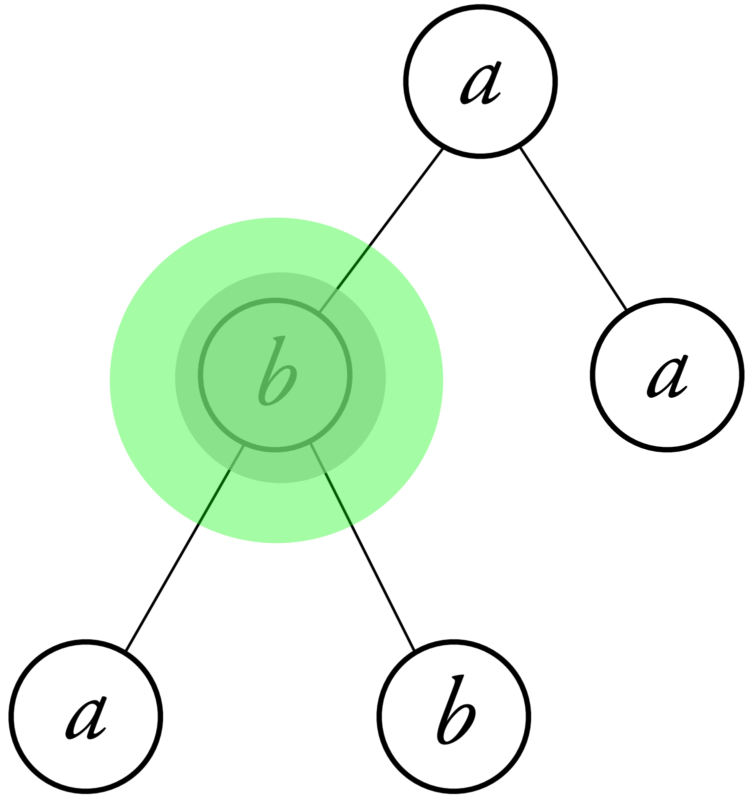
In a large tree with only one type of label, all nodes look the same.
What if the automaton could mark nodes with pebbles?



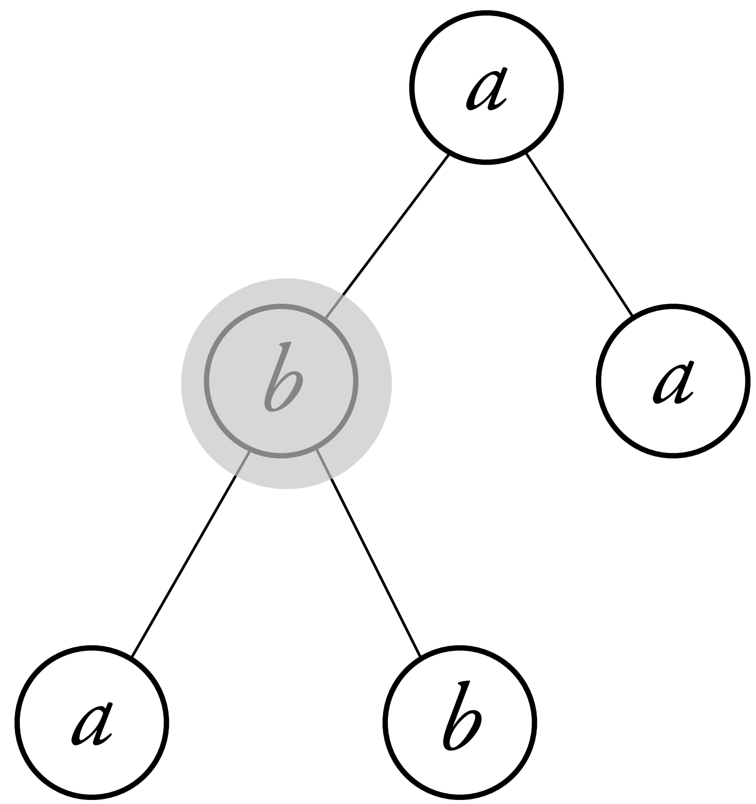
In a large tree with only one type of label, all nodes look the same.
What if the automaton could mark nodes with pebbles?



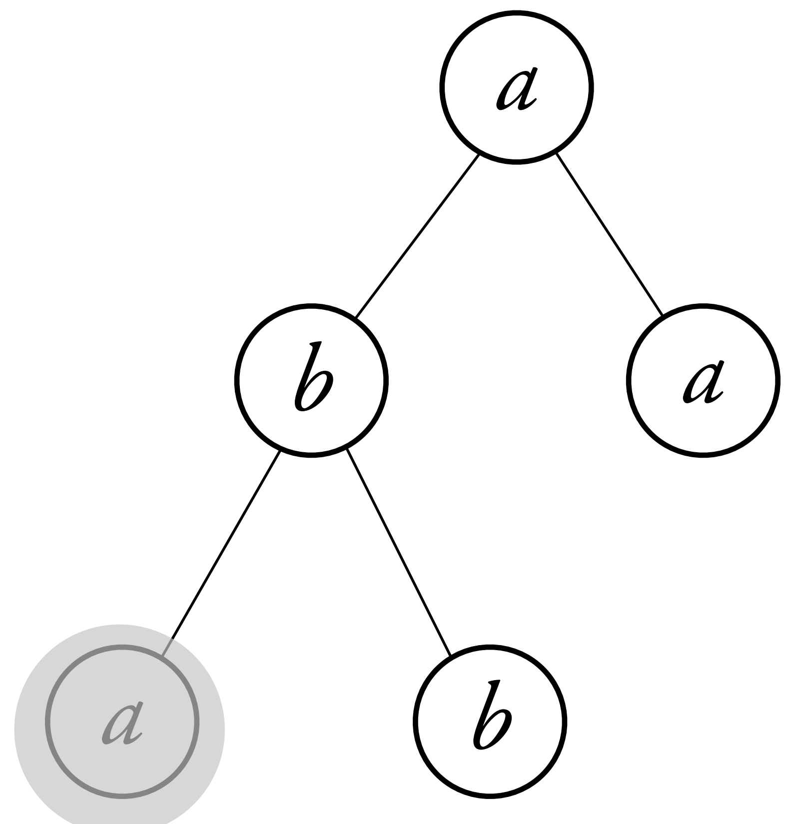
In a large tree with only one type of label, all nodes look the same.
What if the automaton could mark nodes with pebbles?



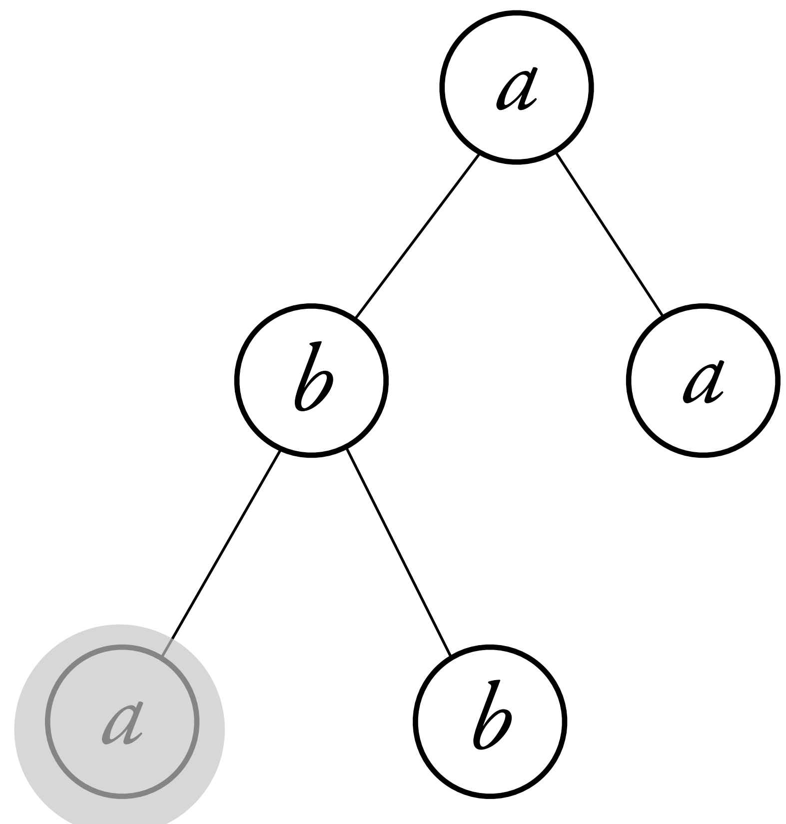
In a large tree with only one type of label, all nodes look the same.
What if the automaton could mark nodes with pebbles?



In a large tree with only one type of label, all nodes look the same. What if the automaton could mark nodes with pebbles?



In a large tree with only one type of label, all nodes look the same.
What if the automaton could mark nodes with pebbles?



An *n-pebble automaton* has pebbles *1,…,n*.
New tests: “is pebble *i* on the current node?”
New commands: “place pebble *i* on the current node”
“lift pebble *i* from the current node”.

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

a b a b a   *a b a b a*

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

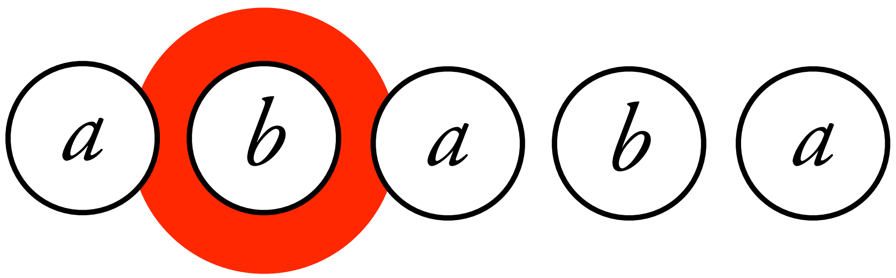
Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

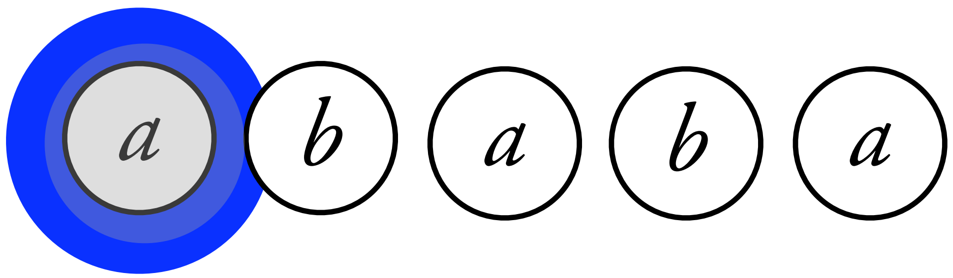
Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

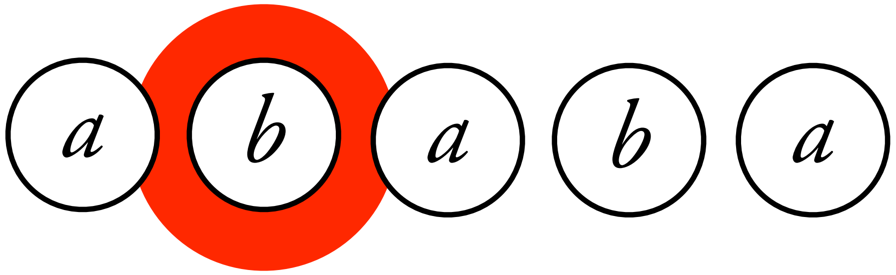
Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

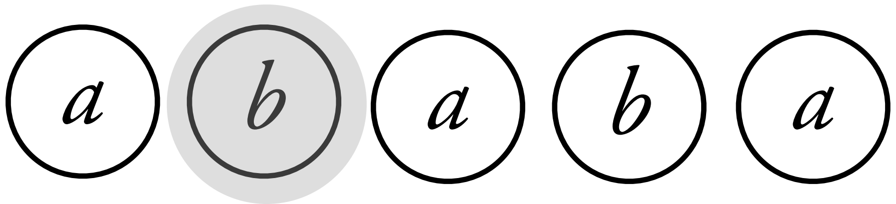




Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

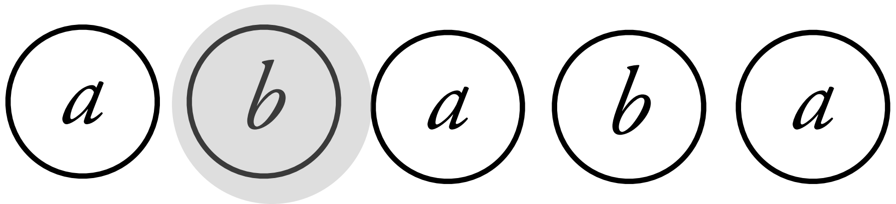


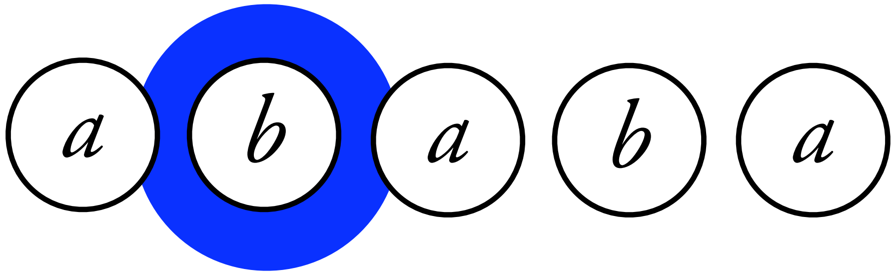


Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

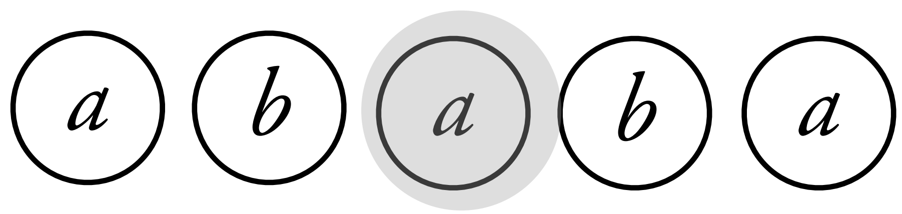
Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

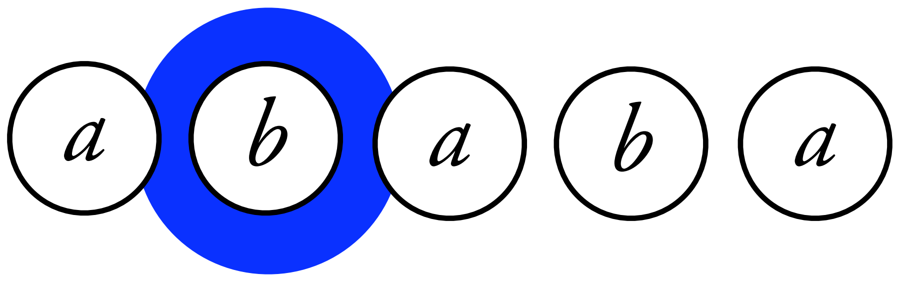
Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.





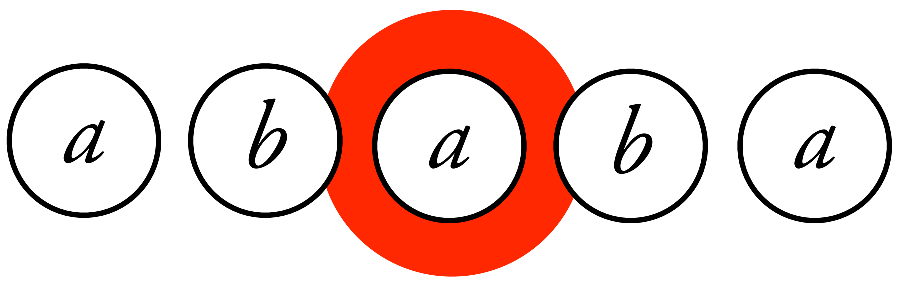
Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

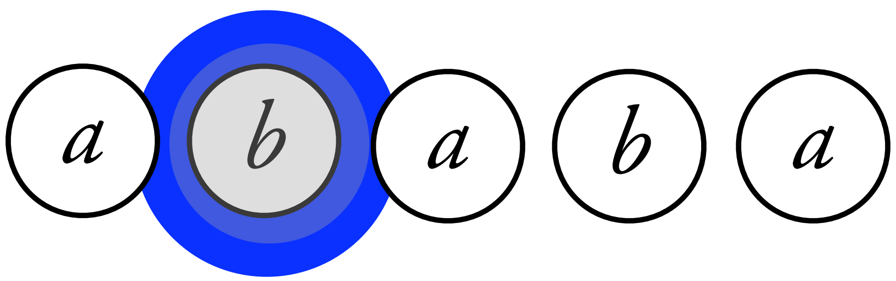




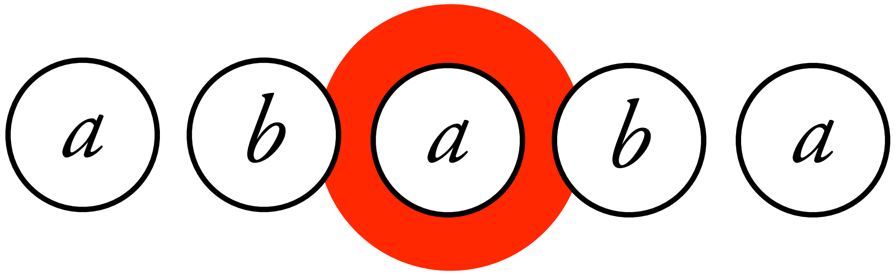
Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

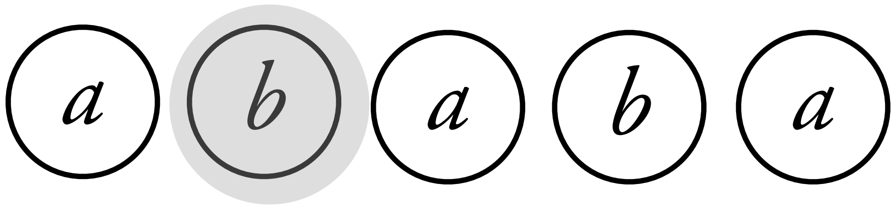
Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.



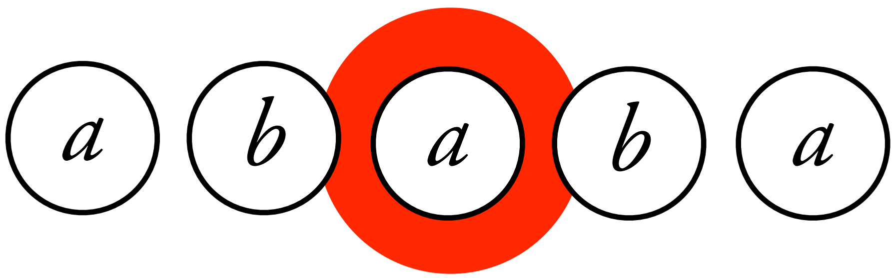


Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.





Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

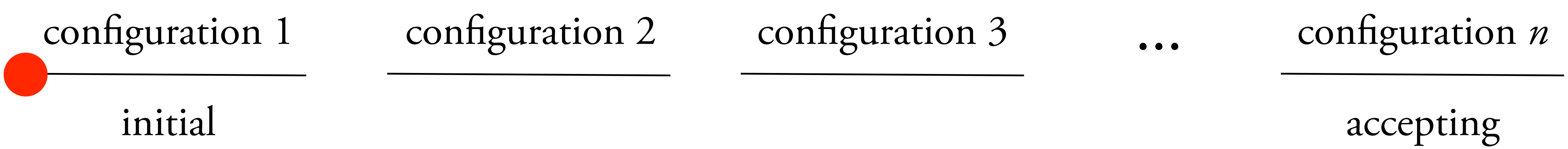


Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

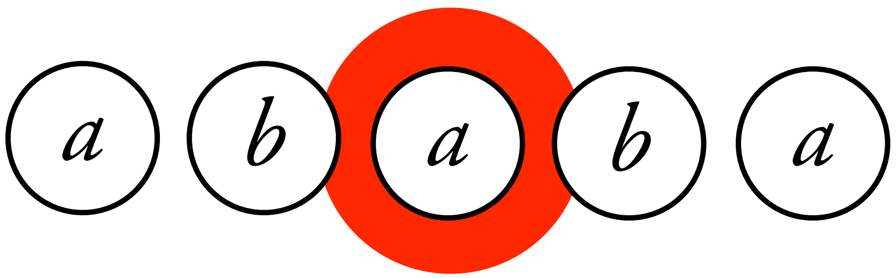
Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

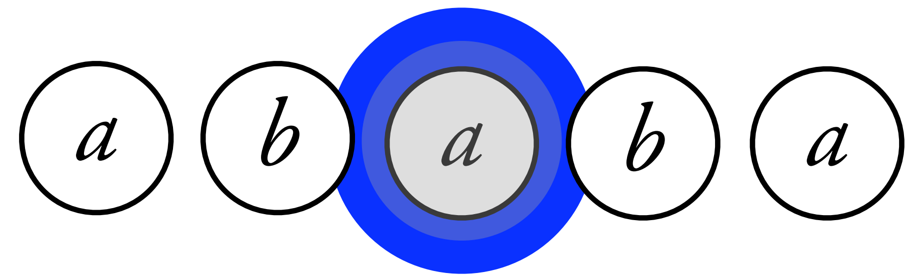
configuration 1	configuration 2	configuration 3	...	configuration n
initial				accepting

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.



Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.





configuration 1 — initial

configuration 2

configuration 3

...

configuration n — accepting

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

configuration 1 — initial

configuration 2

configuration 3

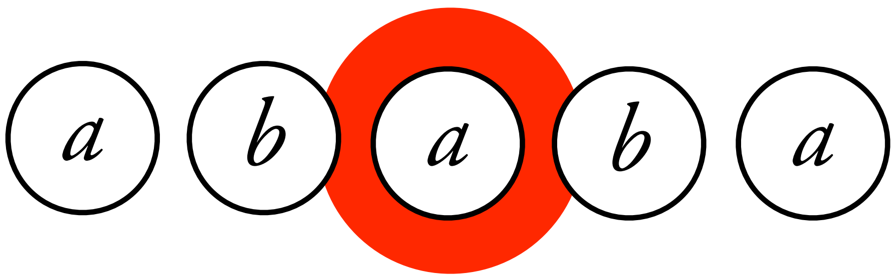
...

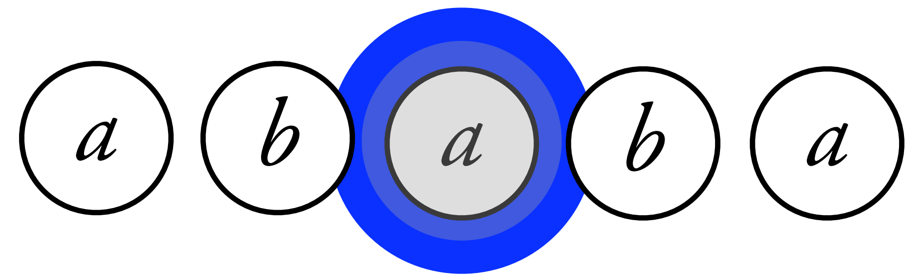
configuration n — accepting

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

configuration 1	configuration 2	configuration 3	...	configuration n
initial				accepting

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.





configuration 1	configuration 2	configuration 3	…	configuration n
initial				accepting

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

configuration 1	configuration 2	configuration 3	…	configuration n
initial				accepting

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

configuration 1 — initial

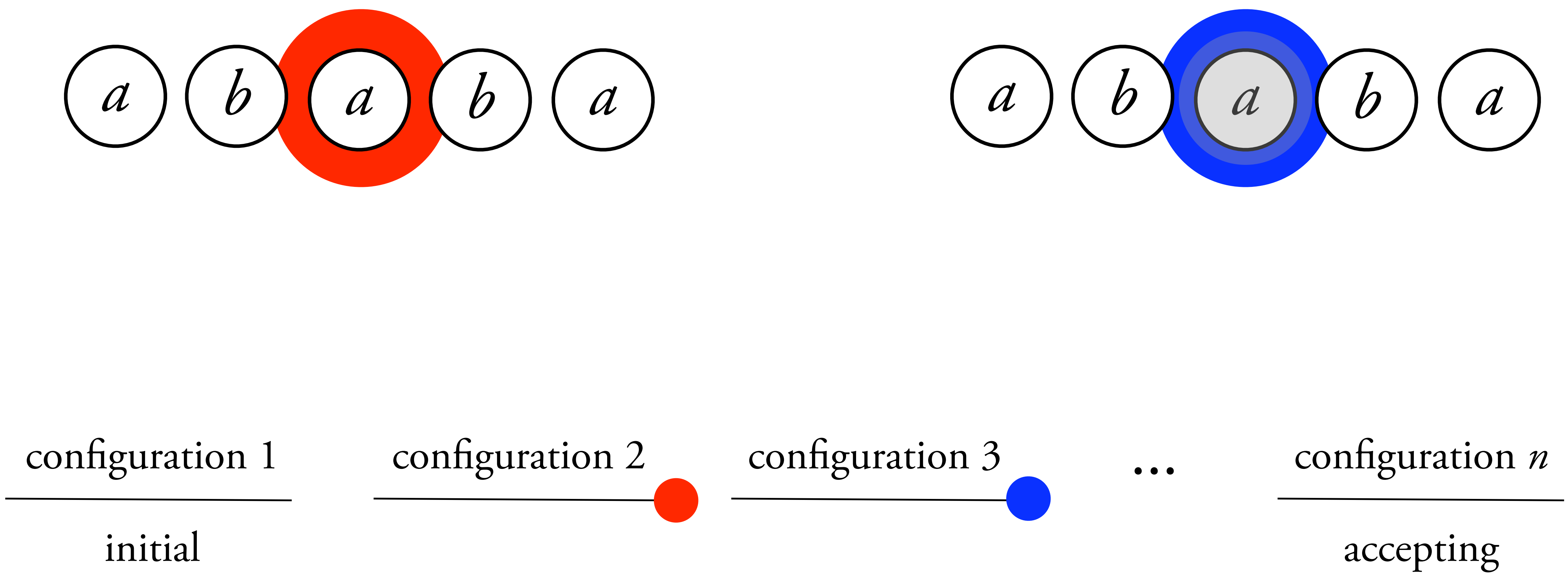
configuration 2

configuration 3

…

configuration n — accepting

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.



Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.

configuration 1 — initial

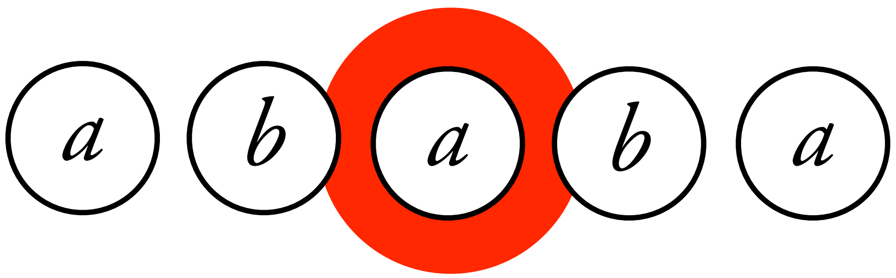
configuration 2

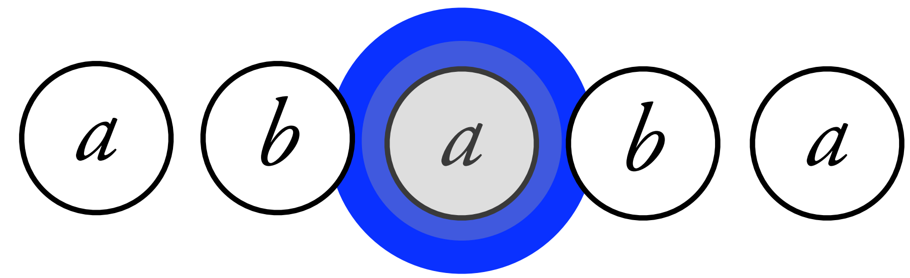
configuration 3

...

configuration n — accepting

Bad news: emptiness is undecidable for pebble automata, even on words and with two pebbles.





configuration 1	configuration 2	configuration 3	...	configuration n
initial				accepting

Good news: with stack discipline, pebble automata have decidable emptiness.

Good news: with stack discipline, pebble automata have decidable emptiness.

The set of pebbles on the tree is always a prefix $1,\ldots,k$ of $1,\ldots,n$. When the newest pebble is i, only i can be lifted, and $i+1$ placed.

Good news: with stack discipline, pebble automata have decidable emptiness.

The set of pebbles on the tree is always a prefix $1,\ldots,k$ of $1,\ldots,n$.
When the newest pebble is i, only i can be lifted, and $i+1$ placed.

Theorem. [Engelfriet, Hoogeboom 99]
Every pebble automaton is equivalent to a tree automaton.

Good news: with stack discipline, pebble automata have decidable emptiness.

The set of pebbles on the tree is always a prefix *1,...,k* of *1,...,n*. When the newest pebble is *i*, only *i* can be lifted, and *i+1* placed.

Theorem. [Engelfriet, Hoogeboom 99]
Every pebble automaton is equivalent to a tree automaton.

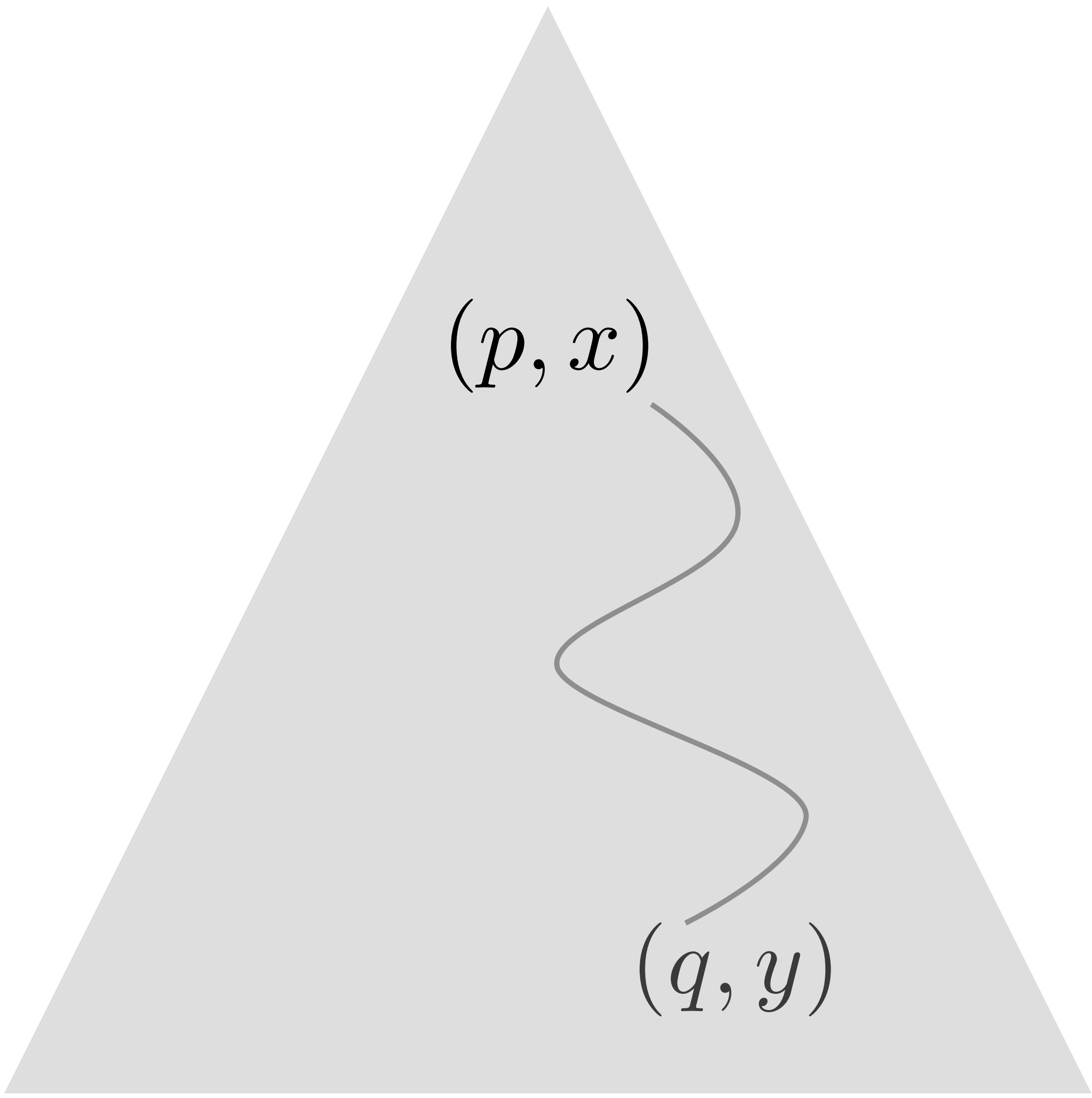
If the pebble automaton has *n* pebbles, the tree automaton may have

$$\overbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}^{n \text{ times}}$$

states. Likewise, emptiness is non-elementary.

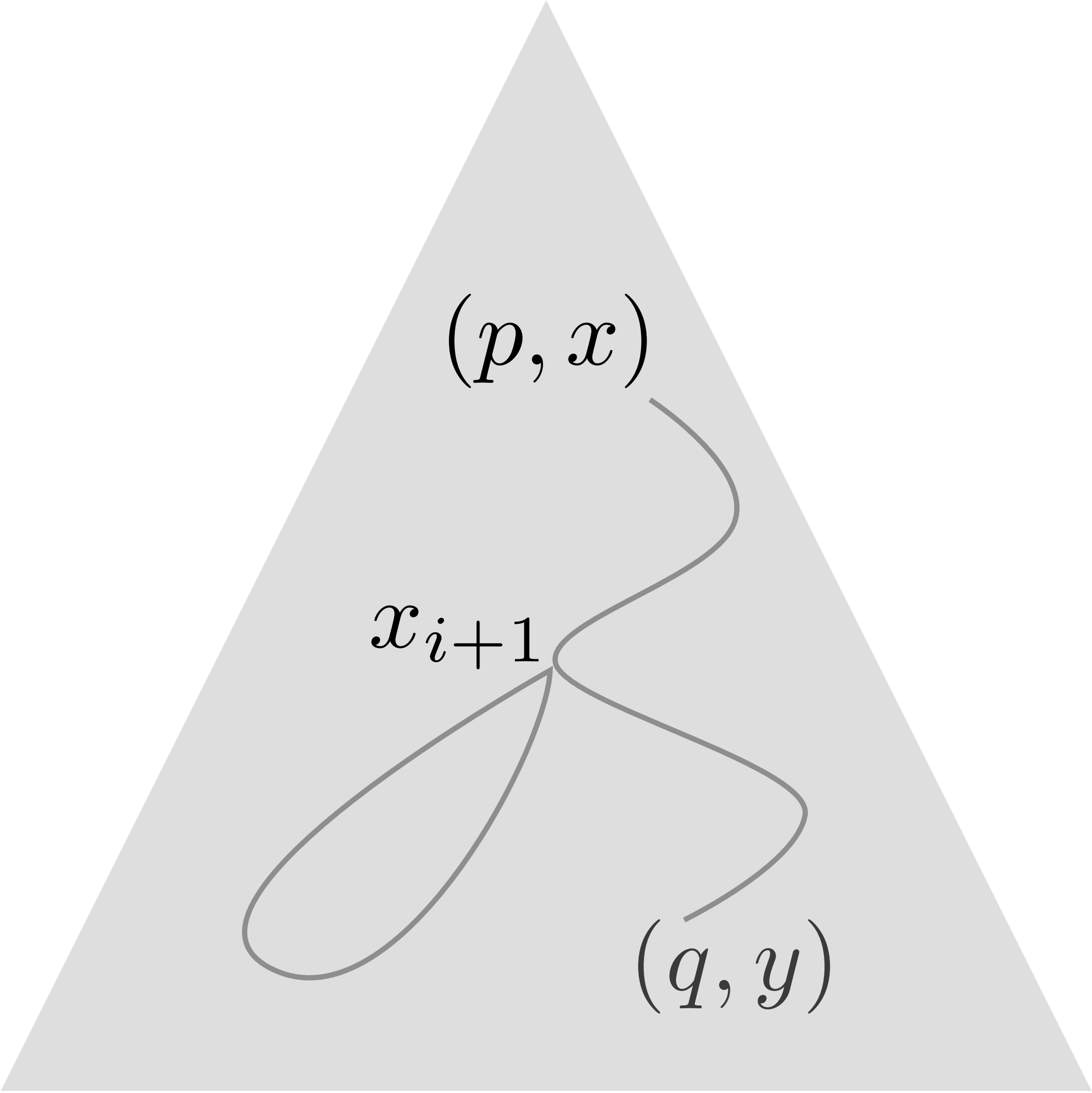
$$\varphi_{p,q}(x, y, x_1, \ldots, x_i)$$

There is a run that begins in (p, x) and ends in (q, y).
The pebbles at the beginning and end are in $x_1, \ldots, x_i$.
During the run, pebble x_i is not lifted, but pebbles can be added.



$$\varphi_{p,q}(x, y, x_1, \ldots, x_i)$$

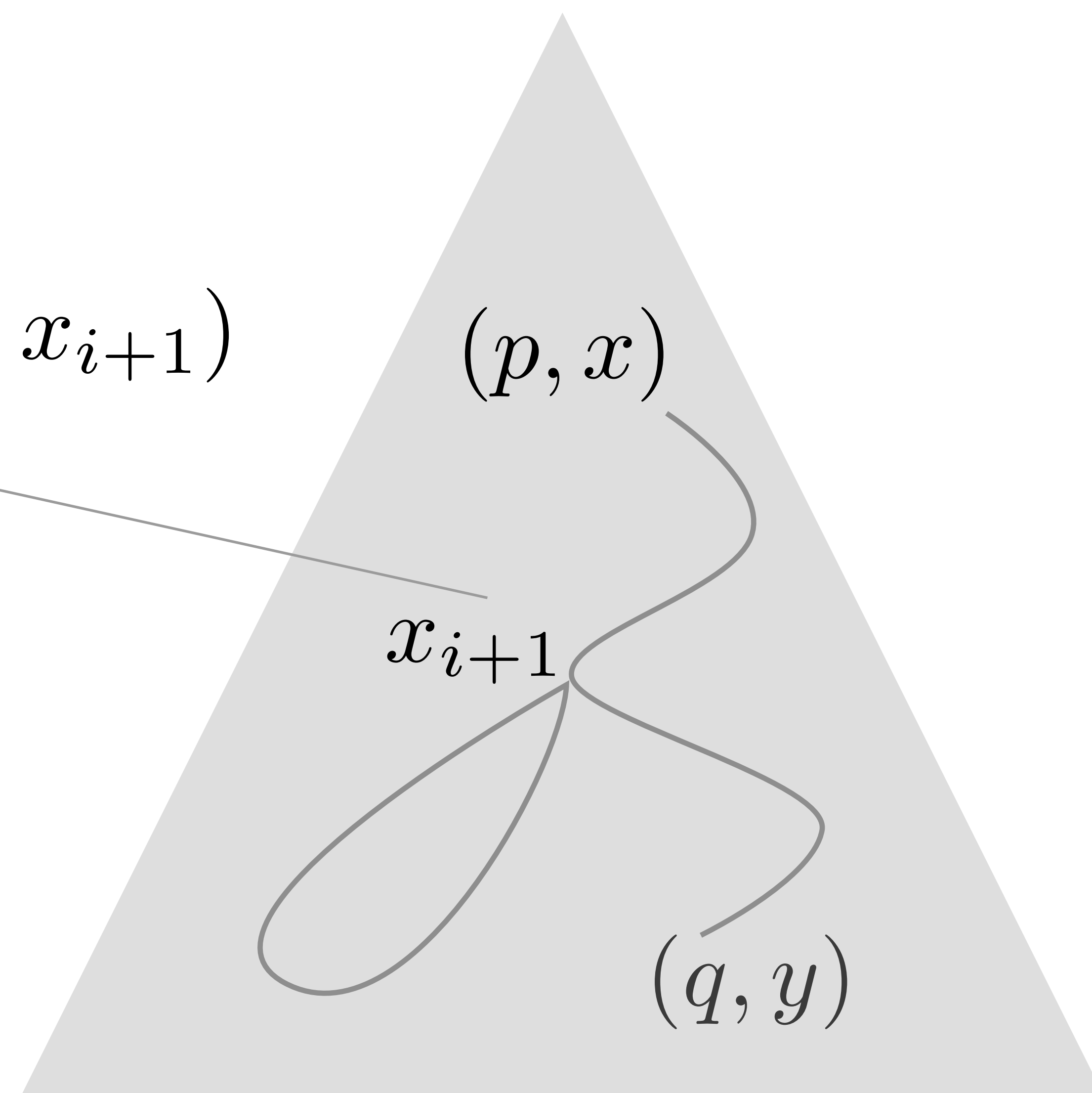
There is a run that begins in (p, x) and ends in (q, y).
The pebbles at the beginning and end are in $x_1, \ldots, x_i$.
During the run, pebble x_i is not lifted, but pebbles can be added.



$$\varphi_{p,q}(x, y, x_1, \ldots, x_i)$$

There is a run that begins in (p, x) and ends in (q, y).
The pebbles at the beginning and end are in $x_1, \ldots, x_i$.
During the run, pebble x_i is not lifted, but pebbles can be added.

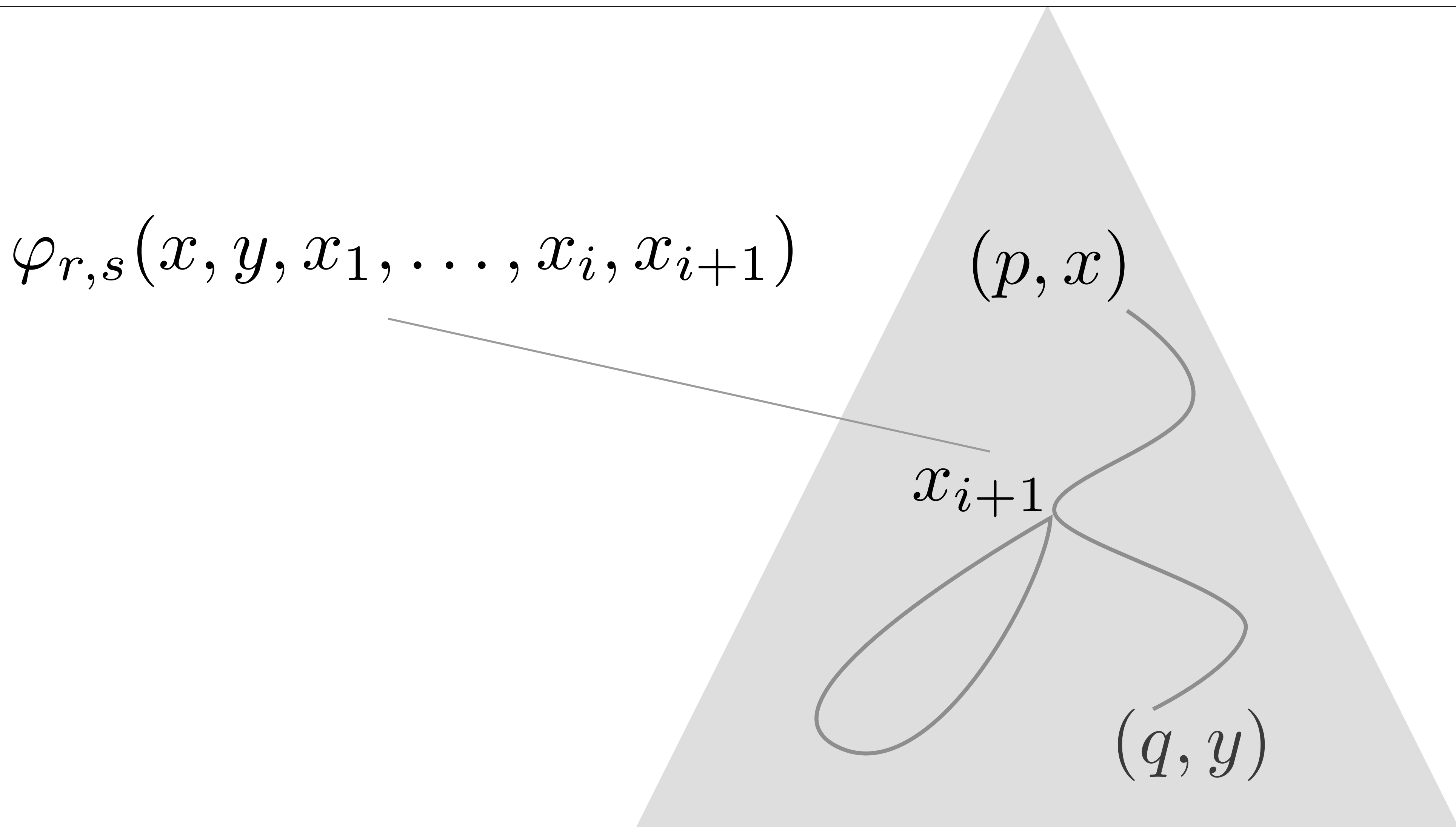
$$\varphi_{r,s}(x, y, x_1, \ldots, x_i, x_{i+1})$$



$$\varphi_{p,q}(x, y, x_1, \ldots, x_i)$$

What logic? Monadic second-order logic is good enough.

Pebble automata = first-order logic with positive transitive closure.

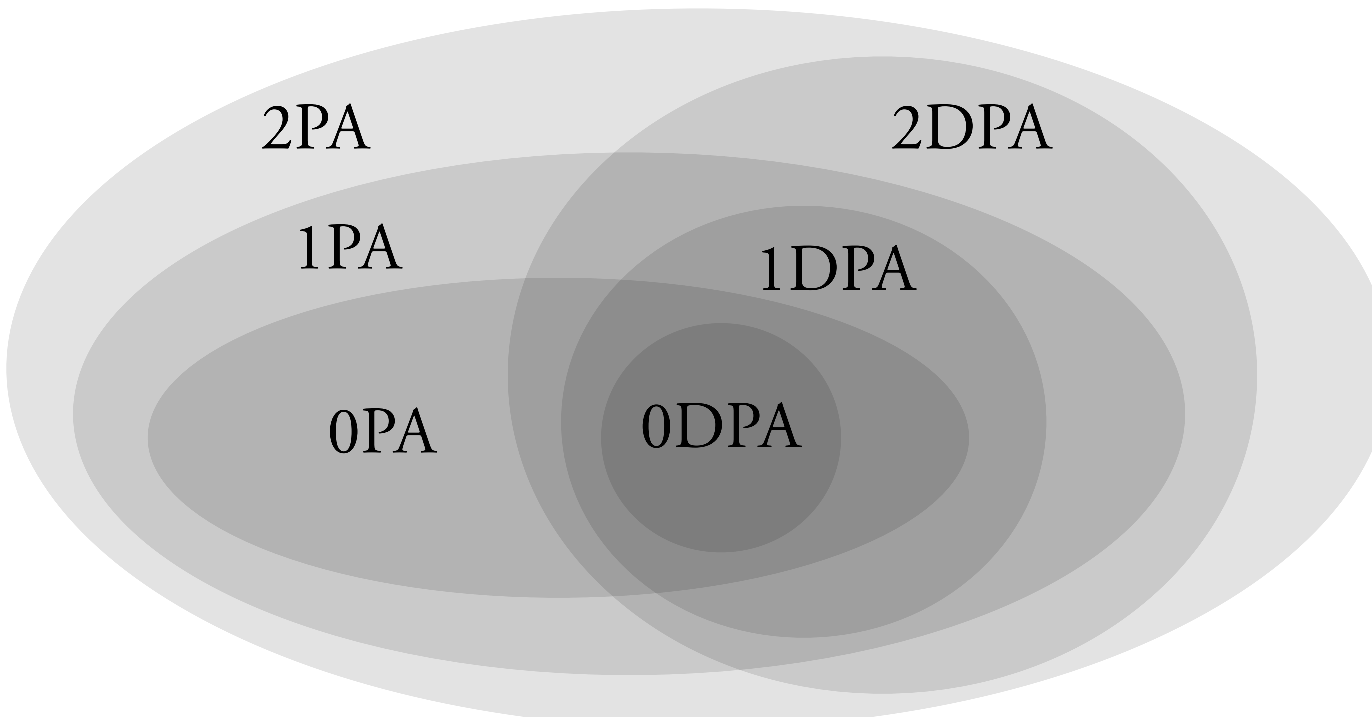


Theorem. [B., Samuelides, Schwentick, Segoufin 06]

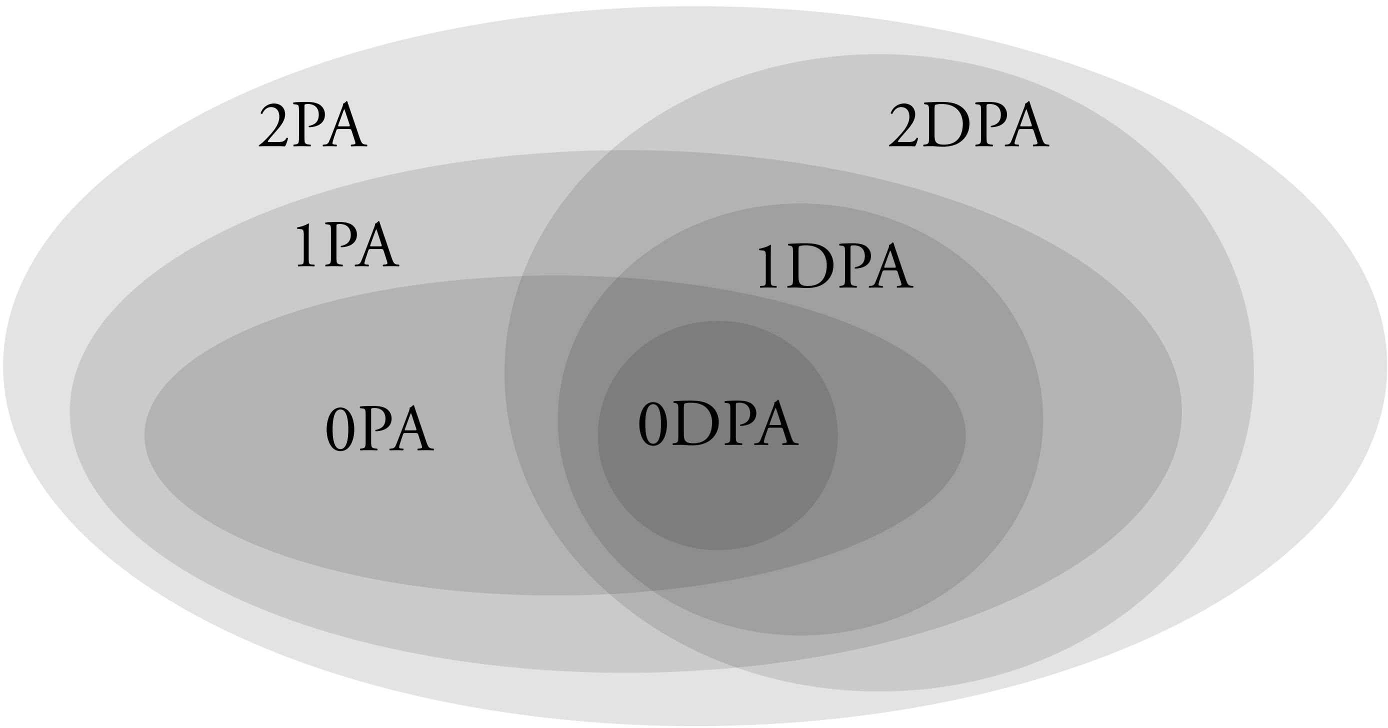
-Pebble automata do not recognize all regular languages.

-Deterministic *n* pebbles are weaker than nondeterminstic *n* pebbles.

-*n* pebbles are weaker than *n+1* pebbles, both in det and nondet.



Theorem. [B., Samuelides, Schwentick, Segoufin 06]
-Pebble automata do not recognize all regular languages.
-Deterministic *n* pebbles are weaker than nondeterminstic *n* pebbles.
-*n* pebbles are weaker than *n+1* pebbles, both in det and nondet.



Open question:

$$\bigcup_i i\mathrm{PA} = \bigcup_i i\mathrm{DPA}$$

Known:

$$\forall i \quad 0\mathrm{PA} \not\subseteq i\mathrm{DPA}$$

Pebble automata = first-order logic with positive transitive closure.

Pebble automata = first-order logic with positive transitive closure.

First-order logic.

$$\forall x \forall y \; a(x) \wedge child(x,y) \Rightarrow b(y)$$

For every nodes *x*, *y*, if *x* has label *a* and *y* is a child of *x*, then *y* has label *b*.

Pebble automata = first-order logic with positive transitive closure.

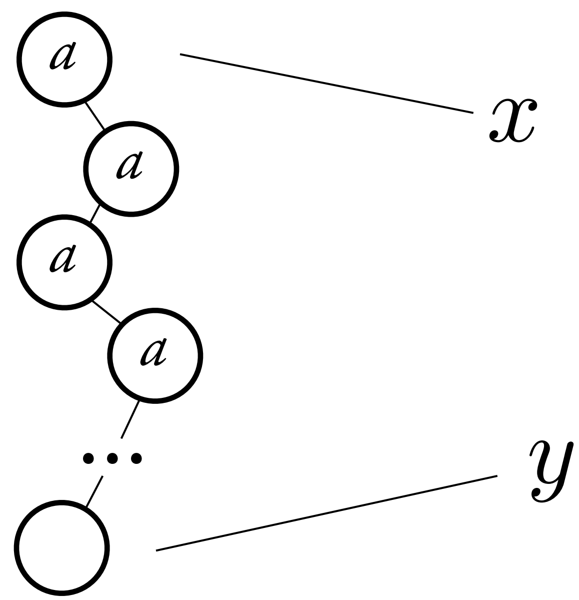
First-order logic.

$$\forall x \forall y \; a(x) \wedge child(x,y) \Rightarrow b(y)$$

For every nodes *x*, *y*, if *x* has label *a* and *y* is a child of *x*, then *y* has label *b*.

First-order logic with transitive closure.

$$TC\big(child(x,y) \wedge a(x)\big)(x,y)$$



For words,
first-order logic with transitive closure = regular languages.

For words,
first-order logic with transitive closure = regular languages.

$$x = y \;\wedge\; a(x)$$

$$(\underline{a + b})^* (aab)^*$$

$$(x = y \;\wedge\; a(x)) \vee (x = y \;\wedge\; b(x))$$

For words,
first-order logic with transitive closure = regular languages.

$$x = y \ \wedge \ a(x)$$

$$(\underline{a + b})^* (aab)^*$$

$$(x = y \ \wedge \ a(x)) \vee (x = y \ \wedge \ b(x))$$

What about trees?
first-order logic with positive transitive closure = pebble automata

For words,
first-order logic with transitive closure = regular languages.

$$x = y \ \wedge \ a(x)$$

$$(\underline{a + b})^*(aab)^*$$

$$(x = y \ \wedge \ a(x)) \vee (x = y \ \wedge \ b(x))$$

What about trees?
first-order logic with positive transitive closure = pebble automata

Theorem. [ten Cate, Segoufin '08]
For trees, not all regular languages are captured by first-order logic with transitive closure.

Conclusion

Conclusion

What did we miss?

Conclusion

What did we miss?
-caterpillar expressions

Conclusion

What did we miss?
-caterpillar expressions
-invisible pebbles

Conclusion

What did we miss?

-caterpillar expressions

-invisible pebbles

-complexity issues

Conclusion

What did we miss?
-caterpillar expressions
-invisible pebbles
-complexity issues

Open questions:

Conclusion

What did we miss?
-caterpillar expressions
-invisible pebbles
-complexity issues

Open questions:
-complementation

Conclusion

What did we miss?
-caterpillar expressions
-invisible pebbles
-complexity issues

Open questions:
-complementation
-deteremінization of pebble automata

Conclusion

What did we miss?
-caterpillar expressions
-invisible pebbles
-complexity issues

Open questions:
-complementation
-deteremінization of pebble automata
-better understanding