Limited Automata and Unary Languages

Giovanni Pighizzini and Luca Prigioniero

Dipartimento di Informatica
Università degli Studi di Milano, Italy

DLT 2017 – Liège
August 7-11, 2017

UNIVERSITÀ DEGLI STUDI
DI MILANO

Limited Automata [Hibbard '67]

One-tape Turing machines with restricted rewritings

Definition

Fixed an integer $d \geq 1$, a *d-limited automaton* is

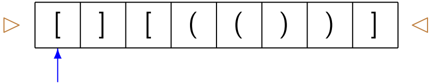
- a one-tape Turing machine
- which is allowed to rewrite the content of each tape cell *only in the first d visits*

Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

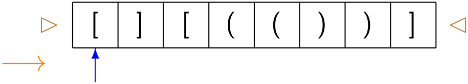


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

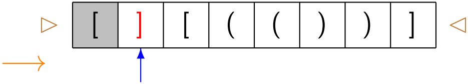


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

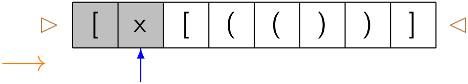


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

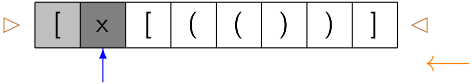


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

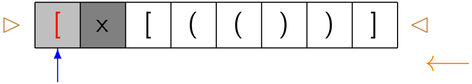


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

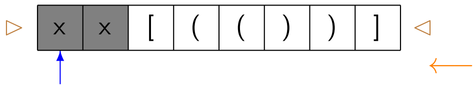
  If it is of the same type, then rewrite it and repeat



Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.
  If it is of the same type, then rewrite it and repeat

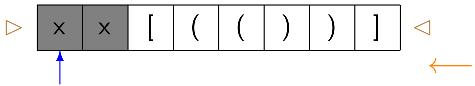


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

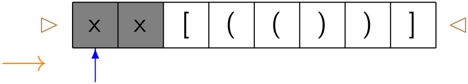


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

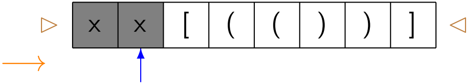


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

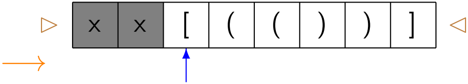


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

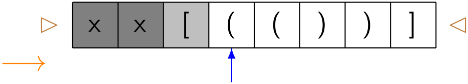


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

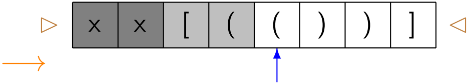


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

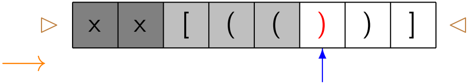


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

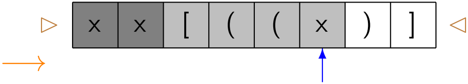


Example: 2-LA for the Dyck Language over $\{[],()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

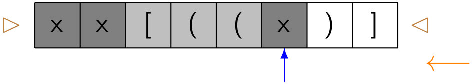


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

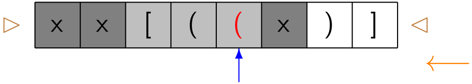


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

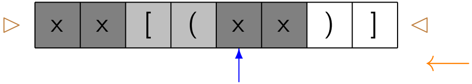


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

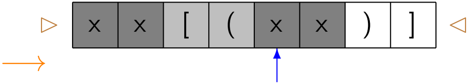


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

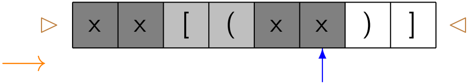


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

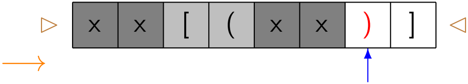


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

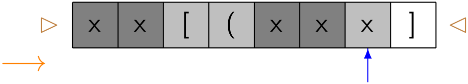
  If it is of the same type, then rewrite it and repeat



Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.
  If it is of the same type, then rewrite it and repeat

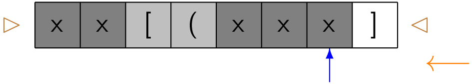


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

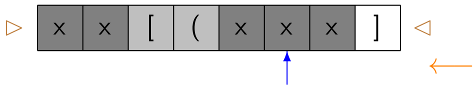


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

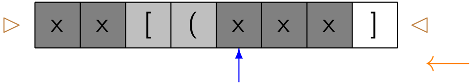


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

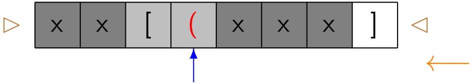


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

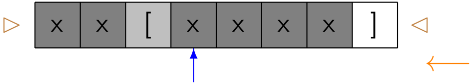


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

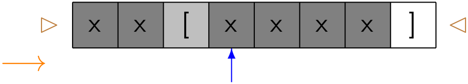


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

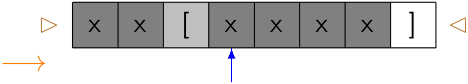


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

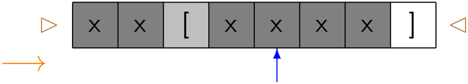


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

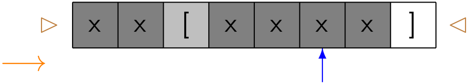


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

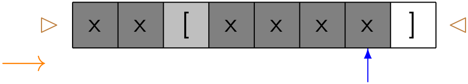


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

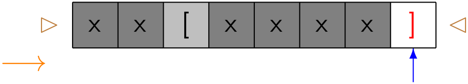


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

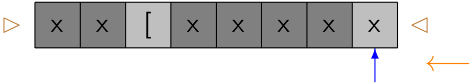


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

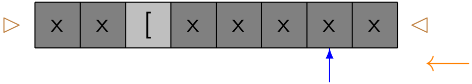


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

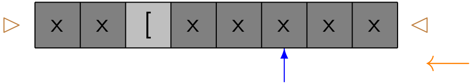


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

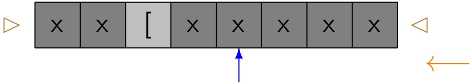


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

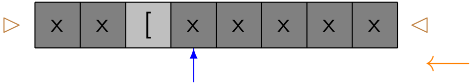


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

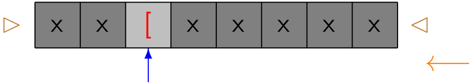


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

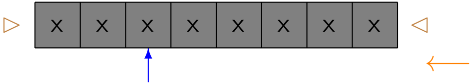


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

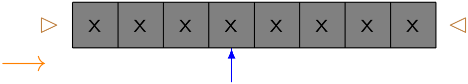


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

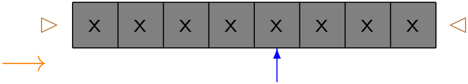


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

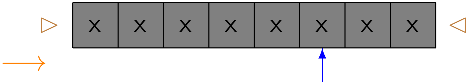


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

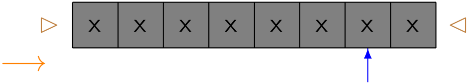


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

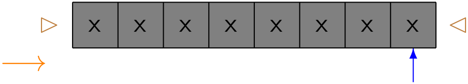


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

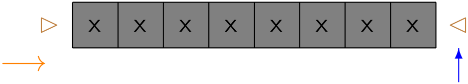


Example: 2-LA for the Dyck Language over $\{[],()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

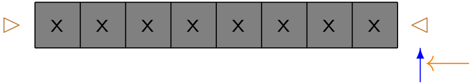


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

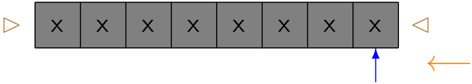


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

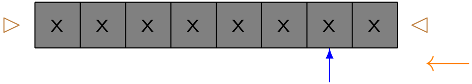


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

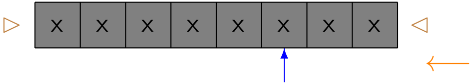


Example: 2-LA for the Dyck Language over $\{[],()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

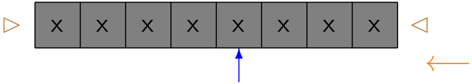


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

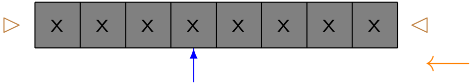


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

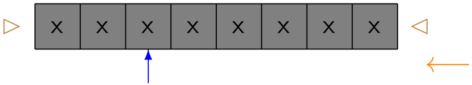


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

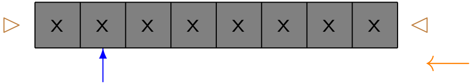


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

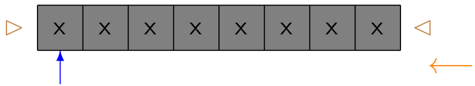


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat

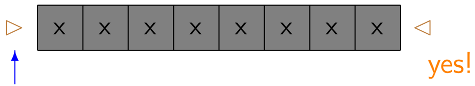


Example: 2-LA for the Dyck Language over $\{[], ()\}$

Idea:

- Move to the right to search a closed bracket and rewrite it
- Then move to the left, to search an open bracket.

  If it is of the same type, then rewrite it and repeat



Each cell is rewritten only in the first 2 visits!

Limited Automata [Hibbard '67]

One-tape Turing machines with restricted rewritings

Definition

Fixed an integer $d \geq 1$, a *d-limited automaton* is

- a one-tape Turing machine
- which is allowed to rewrite the content of each tape cell *only in the first d visits*

Limited Automata [Hibbard '67]

One-tape Turing machines with restricted rewritings

Definition

Fixed an integer $d \geq 1$, a *d-limited automaton* is

- a one-tape Turing machine
- which is allowed to rewrite the content of each tape cell *only in the first d visits*

Computational power

- For each $d \geq 2$, d-limited automata characterize context-free languages [Hibbard '67]
- 1-limited automata characterize regular languages [Wagner&Wechsung '86]

Descriptional Complexity: Limited Automata vs PDAs

- $d = 2$ [P.&Pisoni '15]

Descriptional Complexity: Limited Automata vs PDAs

- $d = 2$ [P.&Pisoni '15]

2-LAs $\rightarrow$ PDAs

Exponential gap

Descriptional Complexity: Limited Automata vs PDAs

- $d = 2$ [P.&Pisoni '15]

2-LAs $\rightarrow$ PDAs

Exponential gap

PDAs $\rightarrow$ 2-LAs

Polynomial upper bound

Descriptional Complexity: Limited Automata vs PDAs

- $d = 2$ [P.&Pisoni '15]

2-LAs $\rightarrow$ PDAs

Exponential gap

PDAs $\rightarrow$ 2-LAs

Polynomial upper bound

- $d > 2$ [Kutrib&P.&Wendlandt *to app.*]

d-LAs $\rightarrow$ PDAs

Still exponential!

Descriptional Complexity: Limited vs Finite Automata

- $d = 1$ [P.&Pisoni '14]

n-state 1-LAs $\rightarrow$ finite automata

	DFA	NFA
nondet. 1-LA		
det. 1-LA		

- The gaps are optimal! (binary witness)

What about the unary case?

Descriptional Complexity: Limited vs Finite Automata

- $d = 1$ [P.&Pisoni '14]

n-state 1-LAs → finite automata	DFA	NFA
nondet. 1-LA	$2^{n \cdot 2^{n^2}}$	
det. 1-LA		

- The gaps are optimal! (binary witness)

What about the unary case?

Descriptional Complexity: Limited vs Finite Automata

- $d = 1$ [P.&Pisoni '14]

n-state 1-LAs $\rightarrow$ finite automata

	DFA	NFA
nondet. 1-LA	$2^{n \cdot 2^{n^2}}$	$n \cdot 2^{n^2}$
det. 1-LA		

- The gaps are optimal! (binary witness)

What about the unary case?

Descriptional Complexity: Limited vs Finite Automata

- $d = 1$ [P.&Pisoni '14]

n-state 1-LAs $\rightarrow$ finite automata

	DFA	NFA
nondet. 1-LA	$2^{n \cdot 2^{n^2}}$	$n \cdot 2^{n^2}$
det. 1-LA	$n \cdot (n+1)^n$	$n \cdot (n+1)^n$

- The gaps are optimal! (binary witness)

What about the unary case?

Descriptional Complexity: Limited vs Finite Automata

- $d = 1$ [P.&Pisoni '14]

n-state 1-LAs $\rightarrow$ finite automata

	DFA	NFA
nondet. 1-LA	$2^{n \cdot 2^{n^2}}$	$n \cdot 2^{n^2}$
det. 1-LA	$n \cdot (n+1)^n$	$n \cdot (n+1)^n$

- The gaps are optimal! (binary witness)

What about the unary case?

Descriptional Complexity: Limited vs Finite Automata

- $d = 1$ [P.&Pisoni '14]

n-state 1-LAs → finite automata

	DFA	NFA
nondet. 1-LA	$2^{n \cdot 2^{n^2}}$	$n \cdot 2^{n^2}$
det. 1-LA	$n \cdot (n+1)^n$	$n \cdot (n+1)^n$

- The gaps are optimal! (binary witness)

What about the unary case?

The Unary Case, $d = 1$

Theorem ([P.&Pisoni '14])

For n prime, the language $\{a^{n^2}\}^*$*:*

- *is accepted by a* 1-LA *with* $n + 1$ *states and a constant size tape alphabet*
- *requires* n^2 *many states to be accepted be a 2NFA*

$\Rightarrow$ Quadratic lower bound for the simulation of unary 1-LAs by finite automata

The Unary Case, $d = 1$

Theorem ([Kutrib&Wendlandt '15])

For n prime, the language $\{a^{n \cdot F(n)}\}$:

- *is accepted by a 1-LA with $4n$ states and a tape alphabet with $n + 1$ symbols*
- *requires $n \cdot F(n)$ many states to be accepted be a 2NFA*

where $F(n) = e^{\sqrt{n \cdot \ln(n)}(1+o(1))}$ (Landau function)

$\Rightarrow$ Superpolynomial lower bound for the simulation of unary 1-LAs by finite automata

The Unary Case, $d = 1$

Theorem ([Kutrib&Wendlandt '15])

For n prime, the language $\{a^{n \cdot F(n)}\}$:

- *is accepted by a 1-LA with $4n$ states and a tape alphabet with $n+1$ symbols*
- *requires $n \cdot F(n)$ many states to be accepted be a 2NFA*

where $F(n) = e^{\sqrt{n \cdot \ln(n)}(1+o(1))}$ (Landau function)

$\Rightarrow$ Superpolynomial lower bound for the simulation of unary 1-LAs by finite automata

This paper: Exponential lower bound

The Unary Case, $d > 1$

- d-LA $\equiv$ CFLs ($d > 1$)
- Each unary CFL is regular [Ginsburg&Rice '62]

$\Rightarrow$ unary d-LA $\equiv$ unary REG

The Unary Case, $d > 1$

- d-LA $\equiv$ CFLs ($d > 1$)
- Each unary CFL is regular [Ginsburg&Rice '62]

$\Rightarrow$ unary d-LA $\equiv$ unary REG

Theorem ([P.&Pisoni '15])

For $n > 0$, the language $\{a^{2^n}\}^*$:

- *is accepted by a deterministic 2-LA of size $O(n)$*
- *requires 2^n many states to be accepted by a 2NFA*

$\Rightarrow$ Exponential lower bound for the simulation of unary 2-LAs by finite automata

The Unary Case, $d > 1$

- d-LA $\equiv$ CFLs ($d > 1$)
- Each unary CFL is regular [Ginsburg&Rice '62]

$\Rightarrow$ unary d-LA $\equiv$ unary REG

Theorem ([P.&Pisoni '15])

For $n > 0$, the language $\{a^{2^n}\}^*$:

- *is accepted by a deterministic 2-LA of size $O(n)$*
- *requires 2^n many states to be accepted by a 2NFA*

$\Rightarrow$ Exponential lower bound for the simulation of unary 2-LAs by finite automata

This paper: Same lower bound for the simulation of unary 1-LAs

Unary 1-LA vs Finite Automata

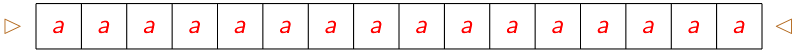
The Exponential Separation

The Witness Language

- Fixed $n > 0$: $L_n = \{a^{2^n}\}$
- The smallest NFA accepting L_n has $2^n + 1$ many states
- We show the existence of a *deterministic* 1-LA of $O(n)$ size accepting L_n

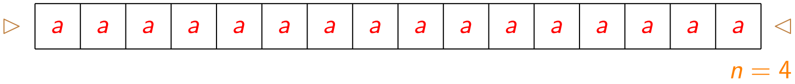
A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: “divide” the input n times by 2



A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: "divide" the input n times by 2



- Make n sweeps of the tape
- At each sweep overwrite each "odd" a

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: "divide" the input n times by 2

▷	X	a	X	a	X	a	X	a	X	a	X	a	X	a	X	a	◁

$n = 4$

- Make n sweeps of the tape
- At each sweep overwrite each "odd" a

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: "divide" the input n times by 2

$\triangleright$	X	X	X	a	X	X	X	a	X	X	X	a	X	X	X	a	$\triangleleft$

$n = 4$

- Make n sweeps of the tape
- At each sweep overwrite each "odd" a

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: "divide" the input n times by 2

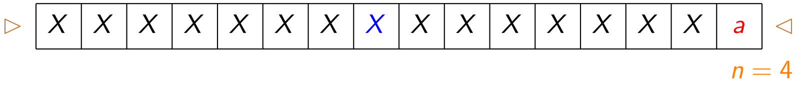
▷	X	X	X	X	X	X	X	a	X	X	X	X	X	X	X	a	◁

$n = 4$

- Make n sweeps of the tape
- At each sweep overwrite each "odd" a

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

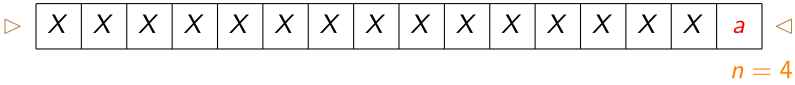
Idea: "divide" the input n times by 2



- Make n sweeps of the tape
- At each sweep overwrite each "odd" a
- Accept if only one a is left on the tape

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: “divide” the input n times by 2



- Make n sweeps of the tape
- At each sweep overwrite each “odd” a
- Accept if only one a is left on the tape
- $O(n)$ states

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: “divide” the input n times by 2

$\triangleright$	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	$\triangleleft$

$n = 4$

Possible variation:

- Rewrite input symbols with the number of current sweep

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: "divide" the input n times by 2

▷	0	a	0	a	0	a	0	a	0	a	0	a	0	a	0	a	◁

$n = 4$

Possible variation:

- Rewrite input symbols with the number of current sweep

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: "divide" the input n times by 2

▷	0	1	0	a	0	1	0	a	0	1	0	a	0	1	0	a	◁

$n = 4$

Possible variation:

- Rewrite input symbols with the number of current sweep

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: "divide" the input n times by 2

▷	0	1	0	2	0	1	0	a	0	1	0	2	0	1	0	a	◁

$n = 4$

Possible variation:

- Rewrite input symbols with the number of current sweep

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: "divide" the input n times by 2

▷	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	a	◁

$n = 4$

Possible variation:

- Rewrite input symbols with the number of current sweep

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: "divide" the input n times by 2

▷	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	◁

$n = 4$

Possible variation:

- Rewrite input symbols with the number of current sweep

A Linear Bounded Automaton for $L_n = \{a^{2^n}\}$

Idea: "divide" the input n times by 2

$\triangleright$	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	$\triangleleft$

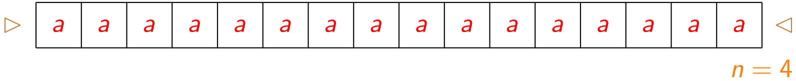
$n = 4$

Possible variation:

- Rewrite input symbols with the number of current sweep

We can build a 1-LA that, for each tape cell,
guesses the number of the sweep
in which this linear bounded automaton rewrites the cell

A 1-Limited Automaton for $L_n = \{a^{2^n}\}$



- *1st sweep:*
  For each cell, guess and write a symbol in $\{0, 1, \ldots, n\}$

A 1-Limited Automaton for $L_n = \{a^{2^n}\}$

$\triangleright$	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	$\triangleleft$

$n = 4$

- *1st sweep:*
  For each cell, guess and write a symbol in $\{0, 1, \ldots, n\}$

A 1-Limited Automaton for $L_n = \{a^{2^n}\}$

▷	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	◁

$n = 4$

- *1st sweep:*
  For each cell, guess and write a symbol in $\{0, 1, \ldots, n\}$
- $(i + 2)$*th sweep,* $i = 0, \ldots, n$*:*
  Verify that the symbol i occurs in all odd positions,
  where positions are counted ignoring cells containing $j < i$

A 1-Limited Automaton for $L_n = \{a^{2^n}\}$

$\triangleright$	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	$\triangleleft$

$n = 4$

- *1st sweep:*
  For each cell, guess and write a symbol in $\{0, 1, \ldots, n\}$
- $(i + 2)$*th sweep,* $i = 0, \ldots, n$*:*
  Verify that the symbol i occurs in all odd positions,
  where positions are counted ignoring cells containing $j < i$

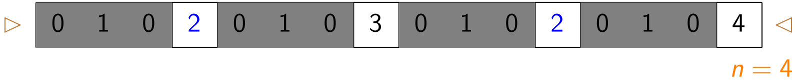
A 1-Limited Automaton for $L_n = \{a^{2^n}\}$

$\triangleright$	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	$\triangleleft$

$n = 4$

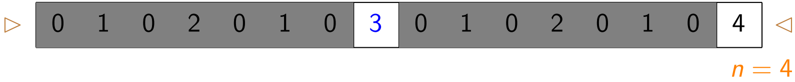
- *1st sweep:*
  For each cell, guess and write a symbol in $\{0, 1, \ldots, n\}$
- $(i + 2)$*th sweep,* $i = 0, \ldots, n$*:*
  Verify that the symbol i occurs in all odd positions,
  where positions are counted ignoring cells containing $j < i$

A 1-Limited Automaton for $L_n = \{a^{2^n}\}$



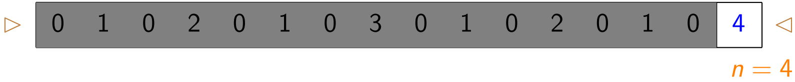
- *1st sweep:*
  For each cell, guess and write a symbol in $\{0, 1, \ldots, n\}$
- $(i + 2)$*th sweep,* $i = 0, \ldots, n$*:*
  Verify that the symbol i occurs in all odd positions,
  where positions are counted ignoring cells containing $j < i$

A 1-Limited Automaton for $L_n = \{a^{2^n}\}$



- *1st sweep:*
  For each cell, guess and write a symbol in $\{0, 1, \ldots, n\}$
- $(i + 2)$*th sweep,* $i = 0, \ldots, n$*:*
  Verify that the symbol i occurs in all odd positions, where positions are counted ignoring cells containing $j < i$

A 1-Limited Automaton for $L_n = \{a^{2^n}\}$



- *1st sweep:*
  For each cell, guess and write a symbol in $\{0, 1, \ldots, n\}$
- $(i + 2)$*th sweep,* $i = 0, \ldots, n$*:*
  Verify that the symbol i occurs in all odd positions,
  where positions are counted ignoring cells containing $j < i$

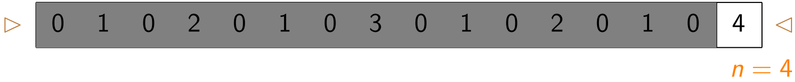
A 1-Limited Automaton for $L_n = \{a^{2^n}\}$

$\triangleright$	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	$\triangleleft$

$n = 4$

- *1st sweep:*
  For each cell, guess and write a symbol in $\{0, 1, \ldots, n\}$
- $(i + 2)$*th sweep,* $i = 0, \ldots, n$*:*
  Verify that the symbol i occurs in all odd positions,
  where positions are counted ignoring cells containing $j < i$
- Size $O(n)$

A 1-Limited Automaton for $L_n = \{a^{2^n}\}$



- *1st sweep:*
  For each cell, guess and write a symbol in $\{0, 1, \ldots, n\}$
- $(i + 2)$*th sweep,* $i = 0, \ldots, n$*:*
  Verify that the symbol i occurs in all odd positions,
  where positions are counted ignoring cells containing $j < i$
- Size $O(n)$

We can do better!

Size $O(n)$, only *deterministic* transitions

The Binary Carry Sequence

The string written by the above linear bounded automaton is a prefix of the *binary carry sequence*:

- First two elements: 0 1
- Next elements: $w \rightarrow ww'$
  - w part already constructed,
  - w' copy of w, with the last symbol replaced by its successor

0 1

The Binary Carry Sequence

The string written by the above linear bounded automaton is a prefix of the *binary carry sequence*:

- First two elements: 0 1
- Next elements: $w \to ww'$
  - w part already constructed,
  - w' copy of w, with the last symbol replaced by its successor

0 1 0 2

The Binary Carry Sequence

The string written by the above linear bounded automaton is a prefix of the *binary carry sequence*:

- First two elements: 0 1
- Next elements: $w \rightarrow ww'$
  - w part already constructed,
  - w' copy of w, with the last symbol replaced by its successor

0 1 0 2 0 1 0 3

The Binary Carry Sequence

The string written by the above linear bounded automaton is a prefix of the *binary carry sequence*:

- First two elements: 0 1
- Next elements: $w \to ww'$
  - w part already constructed,
  - w' copy of w, with the last symbol replaced by its successor

0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4

The Binary Carry Sequence: Properties

- $w_j :=$ prefix of length j of the binary carry sequence
- $BIS(w_j) :=$ *Backward Increasing Sequence* of w_j

  longest increasing sequence obtained with the greedy method by inspecting w_j from the end

$w_{11} = \quad 0 \quad 1 \quad 0 \quad 2 \quad 0 \quad 1 \quad 0 \quad 3 \quad 0 \quad 1 \quad 0$

The Binary Carry Sequence: Properties

- w_j := prefix of length j of the binary carry sequence
- $BIS(w_j)$:= *Backward Increasing Sequence* of w_j

  longest increasing sequence obtained with the greedy method by inspecting w_j from the end

$w_{11} =$ 0 1 0 2 0 1 0 3 0 1 0

$BIS(w_{11}) =$ 0 ...

The Binary Carry Sequence: Properties

- w_j := prefix of length j of the binary carry sequence
- $BIS(w_j)$:= *Backward Increasing Sequence* of w_j

  longest increasing sequence obtained with the greedy method by inspecting w_j from the end

$w_{11} =$ 0 1 0 2 0 1 0 3 0 1 0

$BIS(w_{11}) =$ 0 1 ...

The Binary Carry Sequence: Properties

- w_j := prefix of length j of the binary carry sequence
- $BIS(w_j)$:= *Backward Increasing Sequence* of w_j

  longest increasing sequence obtained with the greedy method by inspecting w_j from the end

$w_{11} =$ 0 1 0 2 0 1 0 3 0 1 0

$BIS(w_{11}) =$ 0 1 3

The Binary Carry Sequence: Properties

- w_j := prefix of length j of the binary carry sequence
- $BIS(w_j)$:= *Backward Increasing Sequence* of w_j

  longest increasing sequence obtained with the greedy method by inspecting w_j from the end

$w_{11} =$ 0 1 0 2 0 1 0 3 0 1 0

$BIS(w_{11}) =$ 0 1 3

$11 = 2^0 + 2^1 + 2^3$

Property 1

$BIS(w_j) =$ positions of 1s in the binary representation of j

The Binary Carry Sequence: Properties

$w_{11} = \quad 0 \quad 1 \quad 0 \quad 2 \quad 0 \quad 1 \quad 0 \quad 3 \quad 0 \quad 1 \quad 0$

$BIS(w_{11}) = \quad 0 \quad 1 \quad 3$

$11 = 2^0 + 2^1 + 2^3$

The Binary Carry Sequence: Properties

$$w_{11} = 0 \quad 1 \quad 0 \quad 2 \quad 0 \quad 1 \quad 0 \quad 3 \quad 0 \quad 1 \quad 0$$

$$BIS(w_{11}) = 0 \quad 1 \quad 3$$

$$11 = 2^0 + 2^1 + 2^3$$

$$12 = 2^2 + 2^3$$

The Binary Carry Sequence: Properties

$$w_{11} = \quad 0 \quad 1 \quad 0 \quad 2 \quad 0 \quad 1 \quad 0 \quad 3 \quad 0 \quad 1 \quad 0$$

$$BIS(w_{11}) = \quad 0 \quad 1 \quad 3$$

$$11 = 2^0 + 2^1 + 2^3$$

$$12 = 2^2 + 2^3$$

$$BIS(w_{12}) = \quad 2 \quad 3$$

The Binary Carry Sequence: Properties

$$w_{11} = \quad 0 \quad 1 \quad 0 \quad 2 \quad 0 \quad 1 \quad 0 \quad 3 \quad 0 \quad 1 \quad 0$$

$$BIS(w_{11}) = \quad 0 \quad 1 \quad 3$$

$$11 = 2^0 + 2^1 + 2^3$$

$$12 = 2^2 + 2^3$$

$$BIS(w_{12}) = \quad 2 \quad 3$$

$$w_{12} = \quad 0 \quad 1 \quad 0 \quad 2 \quad 0 \quad 1 \quad 0 \quad 3 \quad 0 \quad 1 \quad 0 \quad 2$$

The Binary Carry Sequence: Properties

$w_{11} = \quad 0 \quad 1 \quad 0 \quad 2 \quad 0 \quad 1 \quad 0 \quad 3 \quad 0 \quad 1 \quad 0$

$BIS(w_{11}) = \quad 0 \quad 1 \quad 3$

$11 = 2^0 + 2^1 + 2^3$

$12 = 2^2 + 2^3$

$BIS(w_{12}) = \quad 2 \quad 3$

$w_{12} = \quad 0 \quad 1 \quad 0 \quad 2 \quad 0 \quad 1 \quad 0 \quad 3 \quad 0 \quad 1 \quad 0 \quad 2$

Property 2

The symbol of the binary carry sequence in position $j+1$ is the smallest nonnegative integer that does not occur in $BIS(w_j)$

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

Idea: Write on the tape prefixes of the binary carry sequence

$n = 4$

$\triangleright$	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	$\triangleleft$

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

Idea: Write on the tape prefixes of the binary carry sequence

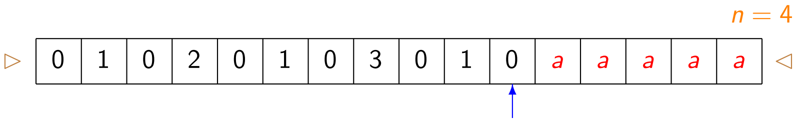
$n = 4$

▷	0	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	◁

- 0 is written on the first cell
- For $j > 0$, with w_j on the first j cells, head on cell j:
  - Compute the smallest $i \notin BIS(w_j)$, inspecting the left part of the tape
  - Move to the right to search the first cell containing a
  - Write i on that cell
- When n is written on a cell:
  - Move one position to the right
  - Accept iff the current cell contains the right endmarker

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

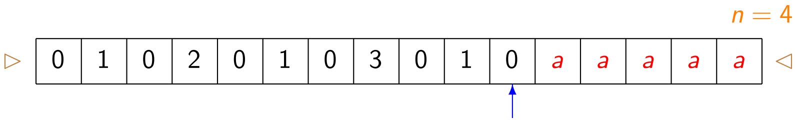
Idea: Write on the tape prefixes of the binary carry sequence



- 0 is written on the first cell
- For $j > 0$, with w_j on the first j cells, head on cell j:
  - Compute the smallest $i \notin BIS(w_j)$, inspecting the left part of the tape
  - Move to the right to search the first cell containing a
  - Write i on that cell
- When n is written on a cell:
  - Move one position to the right
  - Accept iff the current cell contains the right endmarker

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

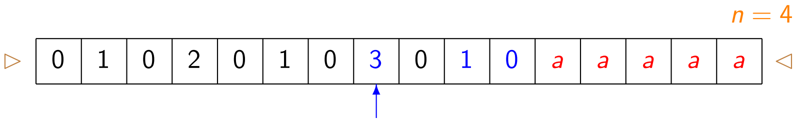
Idea: Write on the tape prefixes of the binary carry sequence



- 0 is written on the first cell
- For $j > 0$, with w_j on the first j cells, head on cell j:
  - Compute the smallest $i \notin BIS(w_j)$, inspecting the left part of the tape
  - Move to the right to search the first cell containing a
  - Write i on that cell
- When n is written on a cell:
  - Move one position to the right
  - Accept iff the current cell contains the right endmarker

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

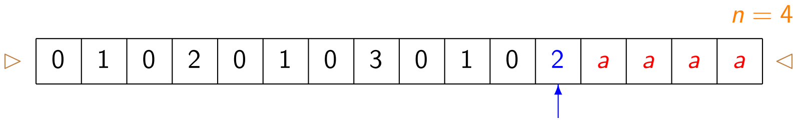
Idea: Write on the tape prefixes of the binary carry sequence



- 0 is written on the first cell
- For $j > 0$, with w_j on the first j cells, head on cell j:
  - Compute the smallest $i \notin BIS(w_j)$, inspecting the left part of the tape
  - Move to the right to search the first cell containing a
  - Write i on that cell
- When n is written on a cell:
  - Move one position to the right
  - Accept iff the current cell contains the right endmarker

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

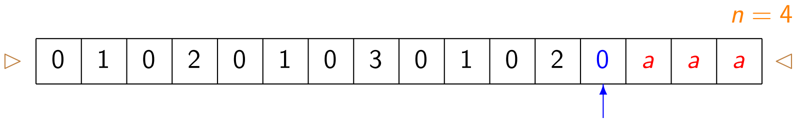
Idea: Write on the tape prefixes of the binary carry sequence



- 0 is written on the first cell
- For $j > 0$, with w_j on the first j cells, head on cell j:
  - Compute the smallest $i \notin BIS(w_j)$, inspecting the left part of the tape
  - Move to the right to search the first cell containing a
  - Write i on that cell
- When n is written on a cell:
  - Move one position to the right
  - Accept iff the current cell contains the right endmarker

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

Idea: Write on the tape prefixes of the binary carry sequence



- 0 is written on the first cell
- For $j > 0$, with w_j on the first j cells, head on cell j:
  - Compute the smallest $i \notin BIS(w_j)$, inspecting the left part of the tape
  - Move to the right to search the first cell containing a
  - Write i on that cell
- When n is written on a cell:
  - Move one position to the right
  - Accept iff the current cell contains the right endmarker

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

Idea: Write on the tape prefixes of the binary carry sequence

$n = 4$

$\triangleright$	0	1	0	2	0	1	0	3	0	1	0	2	0	1	a	a	$\triangleleft$
														↑			

- 0 is written on the first cell
- For $j > 0$, with w_j on the first j cells, head on cell j:
  - Compute the smallest $i \notin BIS(w_j)$, inspecting the left part of the tape
  - Move to the right to search the first cell containing a
  - Write i on that cell
- When n is written on a cell:
  - Move one position to the right
  - Accept iff the current cell contains the right endmarker

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

Idea: Write on the tape prefixes of the binary carry sequence

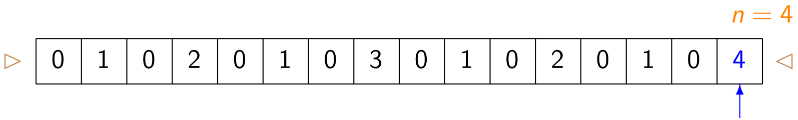
$n = 4$

$\triangleright$	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	a	$\triangleleft$

- 0 is written on the first cell
- For $j > 0$, with w_j on the first j cells, head on cell j:
  - Compute the smallest $i \notin BIS(w_j)$, inspecting the left part of the tape
  - Move to the right to search the first cell containing a
  - Write i on that cell
- When n is written on a cell:
  - Move one position to the right
  - Accept iff the current cell contains the right endmarker

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

Idea: Write on the tape prefixes of the binary carry sequence



- 0 is written on the first cell
- For $j > 0$, with w_j on the first j cells, head on cell j:
  - Compute the smallest $i \notin BIS(w_j)$, inspecting the left part of the tape
  - Move to the right to search the first cell containing a
  - Write i on that cell
- When n is written on a cell:
  - Move one position to the right
  - Accept iff the current cell contains the right endmarker

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

Idea: Write on the tape prefixes of the binary carry sequence

$n = 4$

▷	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	◁

- 0 is written on the first cell
- For $j > 0$, with w_j on the first j cells, head on cell j:
  - Compute the smallest $i \notin BIS(w_j)$, inspecting the left part of the tape
  - Move to the right to search the first cell containing a
  - Write i on that cell
- When n is written on a cell:
  - Move one position to the right
  - Accept iff the current cell contains the right endmarker

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

Idea: Write on the tape prefixes of the binary carry sequence

$n = 4$

$\triangleright$	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	$\triangleleft$

- Each cell is rewritten *only* in the first visit
- Tape alphabet $\{0, \ldots, n\}$
- Finite state control with $O(n)$ states
- Total size of the description $O(n)$

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

Idea: Write on the tape prefixes of the binary carry sequence

$n = 4$

▷	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	◁

- Each cell is rewritten *only* in the first visit
- Tape alphabet $\{0, \ldots, n\}$
- Finite state control with $O(n)$ states
- Total size of the description $O(n)$

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

Idea: Write on the tape prefixes of the binary carry sequence

$n = 4$

▷	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	◁

- Each cell is rewritten *only* in the first visit
- Tape alphabet $\{0, \ldots, n\}$
- Finite state control with $O(n)$ states
- Total size of the description $O(n)$

A Deterministic 1-LA for $L_n = \{a^{2^n}\}$

Idea: Write on the tape prefixes of the binary carry sequence

$n = 4$

$\triangleright$	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	$\triangleleft$

- Each cell is rewritten *only* in the first visit
- Tape alphabet $\{0, \ldots, n\}$
- Finite state control with $O(n)$ states
- Total size of the description $O(n)$

Unary 1-LA vs Finite Automata: Upper and Lower Bounds

det-1-LAs $\rightarrow$ NFAs/DFAs
ndet-1-LAs $\rightarrow$ NFAs

Exponential gap

- l.b. our result
- u.b. general case

Unary 1-LA vs Finite Automata: Upper and Lower Bounds

det-1-LAs $\rightarrow$ NFAs/DFAs
ndet-1-LAs $\rightarrow$ NFAs

Exponential gap

- l.b. our result
- u.b. general case

The gap does not change in the conversion into two-way automata

Unary 1-LA vs Finite Automata: Upper and Lower Bounds

det-1-LAs → NFAs/DFAs
ndet-1-LAs → NFAs

Exponential gap

- l.b. our result
- u.b. general case

The gap does not change in the conversion into two-way automata

ndet-1-LAs → DFAs

- l.b. exp (our result)
- u.b. exp exp (general case)

Unary 1-LA vs Finite Automata: Upper and Lower Bounds

det-1-LAs → NFAs/DFAs
ndet-1-LAs → NFAs

Exponential gap

- l.b. our result
- u.b. general case

The gap does not change in the conversion into two-way automata

ndet-1-LAs → DFAs

- l.b. exp (our result)
- u.b. exp exp (general case)

Problem
Can we reduce the distance between l.b. and u.b.?

From Unary Finite Automata to 1-LAs

An exponential reduction is not always achievable:

Theorem

There is a constant c s.t. for each sufficiently large n there is a unary n-state DFA s.t. all equivalent d-LAs have descriptions of size $> c \cdot n^{1/2}$, for each $d > 0$

Unary CFGs vs Limited Automata

Unary Context-Free Languages

Theorem ([Ginsburg&Rice '62])

Each unary context-free language is regular

Unary Context-Free Languages

Theorem ([Ginsburg&Rice '62])

Each unary context-free language is regular

Theorem ([P.&Shallit&Wang '02])

Each unary context-free grammar can be converted into equivalent DFAs/NFAs of exponential size. These costs cannot be reduced

Unary Context-Free Languages

Theorem ([Ginsburg&Rice '62])

Each unary context-free language is regular

Theorem ([P.&Shallit&Wang '02])

Each unary context-free grammar can be converted into equivalent DFAs/NFAs of exponential size. These costs cannot be reduced

Problem

Study the size relationships between unary CFGs and limited automata

Unary Context-Free Languages

Theorem ([Ginsburg&Rice '62])

Each unary context-free language is regular

Theorem ([P.&Shallit&Wang '02])

Each unary context-free grammar can be converted into equivalent DFAs/NFAs of exponential size. These costs cannot be reduced

Problem

Study the size relationships between unary CFGs and limited automata

[This work]

The conversion
unary CFGs $\rightarrow$ 1-LAs
is polynomial in size

A Variant of the Chomsky-Schützenberger Theorem

Extended Dyck Language $\widehat{D}_\Omega$

- Balanced brackets padded with neutral symbols
- Ex. $\Omega = \{(,), [,], |\}$, strings $||(|)$, $(([|]|)[]||)|()[]|$, ...

A Variant of the Chomsky-Schützenberger Theorem

Extended Dyck Language $\widehat{D}_\Omega$

- Balanced brackets padded with neutral symbols
- Ex. $\Omega = \{(,), [,], |\}$, strings $||(|)$, $(([|]|)[]||)|()[]|$, $\ldots$

Theorem ([Okhotin '12])

$L \subseteq \Sigma^*$ is context-free iff $L = h(\widehat{D}_\Omega \cap R)$, where

- *Ω is an extended bracket alphabet*
- *$R \subseteq \Omega^*$ is regular*
- *$h : \Omega \to \Sigma$ is a letter-to-letter homomorphism*

A Variant of the Chomsky-Schützenberger Theorem

Extended Dyck Language $\widehat{D}_\Omega$

- Balanced brackets padded with neutral symbols
- Ex. $\Omega = \{(,), [,], |\}$, strings $||(|)$, $(([|]|)[]||)|()[]|$, $\dots$

Theorem ([Okhotin '12])

$L \subseteq \Sigma^*$ is context-free iff $L = h(\widehat{D}_\Omega \cap R)$, where

- *Ω is an extended bracket alphabet*
- *$R \subseteq \Omega^*$ is regular*
- *$h : \Omega \to \Sigma$ is a letter-to-letter homomorphism*

Remarks

- The size of Ω is *polynomial* wrt the size of a given CFG G specifying L
- The language R is *local*
- Strings in $\widehat{D}_\Omega \cap R$ encode derivation trees of G

Chomsky-Schützenberger Theorem in the Unary Case

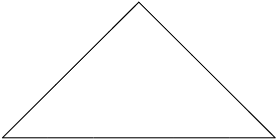
- $G = (V, \{a\}, P, S)$ unary CFG generating $L(G)$
- The membership to $L(G)$ can be witnessed by a *sequence of trees* each one of height $\leq \#V$

Chomsky-Schützenberger Theorem in the Unary Case

- $G = (V, \{a\}, P, S)$ unary CFG generating $L(G)$
- The membership to $L(G)$ can be witnessed by a *sequence of trees* each one of height $\leq \#V$

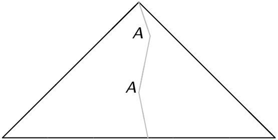
Chomsky-Schützenberger Theorem in the Unary Case

- $G = (V, \{a\}, P, S)$ unary CFG generating $L(G)$
- The membership to $L(G)$ can be witnessed by a *sequence of trees* each one of height $\leq \#V$



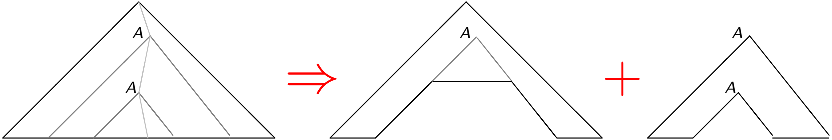
Chomsky-Schützenberger Theorem in the Unary Case

- $G = (V, \{a\}, P, S)$ unary CFG generating $L(G)$
- The membership to $L(G)$ can be witnessed by a *sequence of trees* each one of height $\leq \# V$



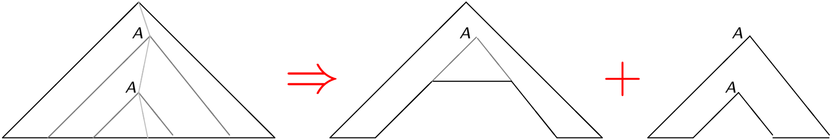
Chomsky-Schützenberger Theorem in the Unary Case

- $G = (V, \{a\}, P, S)$ unary CFG generating $L(G)$
- The membership to $L(G)$ can be witnessed by a *sequence of trees* each one of height $\leq \# V$



Chomsky-Schützenberger Theorem in the Unary Case

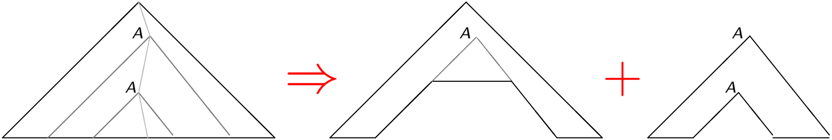
- $G = (V, \{a\}, P, S)$ unary CFG generating $L(G)$
- The membership to $L(G)$ can be witnessed by a *sequence of trees* each one of height $\leq \#V$



Then $L(G) = h(\widehat{D}^{(\#V)}_{\Omega_G} \cap R)$

Chomsky-Schützenberger Theorem in the Unary Case

- $G = (V, \{a\}, P, S)$ unary CFG generating $L(G)$
- The membership to $L(G)$ can be witnessed by a *sequence of trees* each one of height $\leq \#V$



Then $$L(G) = h(\widehat{D}^{(\#V)}_{\Omega_G} \cap R)$$

The "restricted extended" Dyck Language $\widehat{D}^{(\#V)}_{\Omega_G} \subset \widehat{D}_{\Omega_G}$

- contains only the strings with bracket nesting depth $\leq \#V$
- is recognized by a 2DFA of size polynomial wrt the size of G

A 1-LA Accepting $L(G) = h(\widehat{D}_{\Omega_G}^{(\#V)} \cap R)$

1. Input a^m
2. Guess $w \in h^{-1}(a^m)$
   - Scan the tape from left to right
   - Rewrite each input cell with a symbol from Ω_G
3. Check if $w \in \widehat{D}_{\Omega_G}^{(\#V)}$
   - 2DFA of polynomial size
4. Check if $w \in R$
   - DFA of polynomial size

A 1-LA Accepting $L(G) = h(\widehat{D}_{\Omega_G}^{(\#V)} \cap R)$

1. Input a^m
2. Guess $w \in h^{-1}(a^m)$
   - Scan the tape from left to right
   - Rewrite each input cell with a symbol from Ω_G
3. Check if $w \in \widehat{D}_{\Omega_G}^{(\#V)}$
   - 2DFA of polynomial size
4. Check if $w \in R$
   - DFA of polynomial size

A 1-LA Accepting $L(G) = h(\widehat{D}_{\Omega_G}^{(\#V)} \cap R)$

1. Input a^m
2. Guess $w \in h^{-1}(a^m)$
   - Scan the tape from left to right
   - Rewrite each input cell with a symbol from Ω_G
3. Check if $w \in \widehat{D}_{\Omega_G}^{(\#V)}$
   - 2DFA of polynomial size
4. Check if $w \in R$
   - DFA of polynomial size

A 1-LA Accepting $L(G) = h(\widehat{D}_{\Omega_G}^{(\#V)} \cap R)$

1. Input a^m
2. Guess $w \in h^{-1}(a^m)$
   - Scan the tape from left to right
   - Rewrite each input cell with a symbol from Ω_G
3. Check if $w \in \widehat{D}_{\Omega_G}^{(\#V)}$
   - 2DFA of polynomial size
4. Check if $w \in R$
   - DFA of polynomial size

A 1-LA Accepting $L(G) = h(\widehat{D}_{\Omega_G}^{(\#V)} \cap R)$

1. Input a^m
2. Guess $w \in h^{-1}(a^m)$
   - Scan the tape from left to right
   - Rewrite each input cell with a symbol from Ω_G
3. Check if $w \in \widehat{D}_{\Omega_G}^{(\#V)}$
   - 2DFA of polynomial size
4. Check if $w \in R$
   - DFA of polynomial size

Summing up:

- Each cell is rewritten only in the first visit
- The total size of the resulting 1-LA is polynomial

We proved that

Theorem

The conversion of unary CFGs into 1-LAs is polynomial in size

Unary CFGs vs Limited Automata

We proved that

Theorem

The conversion of unary CFGs into 1-LAs is polynomial in size

Problems

- Cost of the converse conversion, i.e., (unary) 1-LAs $\rightarrow$ CFGs
  General alphabets: 2-LAs $\rightarrow$ CFGs is *exponential* in size
- Conversion of unary CFGs into *deterministic* limited automata

Unary CFGs vs Limited Automata

We proved that

Theorem

The conversion of unary CFGs into 1-LAs is polynomial in size

Problems

- Cost of the converse conversion, i.e., (unary) 1-LAs $\rightarrow$ CFGs
  General alphabets: 2-LAs $\rightarrow$ CFGs is *exponential* in size
- Conversion of unary CFGs into *deterministic* limited automata

Thank you for your attention!