Data Structures for Incremental Maintenance of String Properties under Updates

Antoine Amarilli

May 9, 2022

Télécom Paris

Incremental maintenance on strings

The concept: **incremental maintenance**

- You have a **string**:
  - $\rightarrow$ e.g., `aaaabaabaca`

Incremental maintenance on strings

The concept: incremental maintenance

- You have a string:
  - → e.g., `aaaabaabaca`
- You are interested in a property
  - → e.g., having at least one `a`

Incremental maintenance on strings

The concept: incremental maintenance

- You have a string:
  - → e.g., `aaaabaabaca`
- You are interested in a property
  - → e.g., having at least one `a`
- The string is updated
  - → e.g., replace the 3rd character by an `a`

Incremental maintenance on strings

The concept: incremental maintenance

- You have a string:
  → e.g., `aaaabaabaca`

- You are interested in a property
  → e.g., having at least one `a`

- The string is updated
  → e.g., replace the 3rd character by an `a`

- You want to maintain the property efficiently
  → e.g., with low running time or memory overhead

Naive vs efficient algorithms

How can we **efficiently maintain** the property “having at least one `a`” under **substitutions** on an input string w of length n?

- **Naive algorithm:**

Naive vs efficient algorithms

How can we **efficiently maintain** the property "having at least one `a`" under **substitutions** on an input string w of length n?

- **Naive algorithm:** After each substitution, go over w and search for an `a`

Naive vs efficient algorithms

How can we efficiently maintain the property “having at least one `a`” under substitutions on an input string w of length n?

- Naive algorithm: After each substitution, go over w and search for an `a`
  $\rightarrow$ Complexity per update:

Naive vs efficient algorithms

How can we **efficiently maintain** the property "having at least one `a`" under **substitutions** on an input string w of length n?

- **Naive algorithm:** After each substitution, go over w and search for an `a`
  $\rightarrow$ Complexity per update: **linear** in the length of w, i.e., in $O(n)$

Naive vs efficient algorithms

How can we **efficiently maintain** the property "having at least one `a`" under **substitutions** on an input string w of length n?

- **Naive algorithm:** After each substitution, go over w and search for an `a`
  $\rightarrow$ Complexity per update: **linear** in the length of w, i.e., in $O(n)$
- **Clever algorithm:**

Naive vs efficient algorithms

How can we **efficiently maintain** the property “having at least one `a`” under **substitutions** on an input string w of length n?

- **Naive algorithm:** After each substitution, go over w and search for an `a`
  $\rightarrow$ Complexity per update: **linear** in the length of w, i.e., in $O(n)$

- **Clever algorithm:** Maintain a **counter** κ of the number of `a`'s

Naive vs efficient algorithms

How can we **efficiently maintain** the property "having at least one `a`" under **substitutions** on an input string w of length n?

- **Naive algorithm:** After each substitution, go over w and search for an `a`
  $\rightarrow$ Complexity per update: **linear** in the length of w, i.e., in $O(n)$

- **Clever algorithm:** Maintain a **counter** κ of the number of `a`'s
  - If you replace an `a` by another character, decrement κ

Naive vs efficient algorithms

How can we **efficiently maintain** the property “having at least one `a`” under **substitutions** on an input string w of length n?

- **Naive algorithm:** After each substitution, go over w and search for an `a`
  $\rightarrow$ Complexity per update: **linear** in the length of w, i.e., in $O(n)$
- **Clever algorithm:** Maintain a **counter** κ of the number of `a`’s
  - If you replace an `a` by another character, decrement κ
  - If you replace another character by an `a`, increment κ

Naive vs efficient algorithms

How can we **efficiently maintain** the property "having at least one $\texttt{a}$" under **substitutions** on an input string w of length n?

- **Naive algorithm:** After each substitution, go over w and search for an $\texttt{a}$
  $\rightarrow$ Complexity per update: **linear** in the length of w, i.e., in $O(n)$

- **Clever algorithm:** Maintain a **counter** κ of the number of $\texttt{a}$'s
  - If you replace an $\texttt{a}$ by another character, decrement κ
  - If you replace another character by an $\texttt{a}$, increment κ
  - If $\kappa > 0$ then w contains an $\texttt{a}$

Naive vs efficient algorithms

How can we **efficiently maintain** the property "having at least one `a`" under **substitutions** on an input string w of length n?

- **Naive algorithm:** After each substitution, go over w and search for an `a`

  $\rightarrow$ Complexity per update: **linear** in the length of w, i.e., in $O(n)$

- **Clever algorithm:** Maintain a **counter** κ of the number of `a`'s
  - If you replace an `a` by another character, decrement κ
  - If you replace another character by an `a`, increment κ
  - If $\kappa > 0$ then w contains an `a`

  $\rightarrow$ Complexity per update:

Naive vs efficient algorithms

How can we **efficiently maintain** the property "having at least one `a`" under **substitutions** on an input string w of length n?

- **Naive algorithm:** After each substitution, go over w and search for an `a`
  $\rightarrow$ Complexity per update: **linear** in the length of w, i.e., in $O(n)$

- **Clever algorithm:** Maintain a **counter** κ of the number of `a`'s
  - If you replace an `a` by another character, decrement κ
  - If you replace another character by an `a`, increment κ
  - If $\kappa > 0$ then w contains an `a`

  $\rightarrow$ Complexity per update: **constant** (in the RAM model)

Structure of the talk

We focus on the dynamic membership problem:
incremental maintenance of membership to a regular language

- Dynamic membership under substitution updates
  - A general-purpose $O(\log n)$ algorithm
  - Better algorithms for specific languages: [A., Jachiet, Paperman, ICALP'21]

Structure of the talk

We focus on the dynamic membership problem:
incremental maintenance of membership to a regular language

- Dynamic membership under substitution updates
  - A general-purpose $O(\log n)$ algorithm
  - Better algorithms for specific languages: [A., Jachiet, Paperman, ICALP'21]
- Dynamic membership under other update operations
  - Endpoint updates: push and pop at the beginning and end
  - Insertions and deletions
  - Splitting and joining

Structure of the talk

We focus on the dynamic membership problem:
incremental maintenance of membership to a regular language

- Dynamic membership under substitution updates
  - A general-purpose $O(\log n)$ algorithm
  - Better algorithms for specific languages: [A., Jachiet, Paperman, ICALP'21]

- Dynamic membership under other update operations
  - Endpoint updates: push and pop at the beginning and end
  - Insertions and deletions
  - Splitting and joining

- Beyond dynamic membership: incremental maintenance for enumeration

Regular languages and substitution updates

Problem: dynamic membership for regular languages under substitutions

- Fix a regular language L
  - $\rightarrow$ E.g., $L = (ab)^*$
- Read an input string w with $n := |w|$
  - $\rightarrow$ E.g., $w = abbbab$

Problem: dynamic membership for regular languages under substitutions

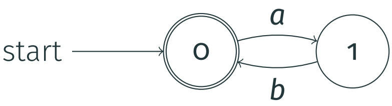
- Fix a **regular language** L
  - → E.g., $L = (ab)^*$
- Read an **input string** w with $n := |w|$
  - → E.g., $w = abbbab$
- **Maintain** the membership of w to L under **substitution updates**
  - → Initially, we have $w \notin L$
  - → Replace character at position 3 with a: we now have $w \in L$
  - → The **length** n never changes

Problem: dynamic membership for regular languages under substitutions

- Fix a regular language L
  - → E.g., $L = (ab)^*$
- Read an input string w with $n := |w|$
  - → E.g., $w = abbbab$
- Maintain the membership of w to L under substitution updates
  - → Initially, we have $w \notin L$
  - → Replace character at position 3 with a: we now have $w \in L$
  - → The length n never changes
- Model: RAM model
  - Cell size in $\Theta(\log(n))$
  - Unit-cost arithmetics

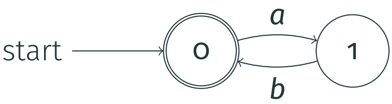
A general-purpose $O(\log n)$ algorithm

Fix the language $L = (ab)^*$:



A general-purpose $O(\log n)$ algorithm

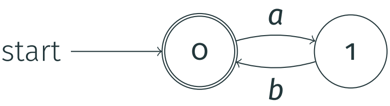
Fix the language $L = (ab)^*$:



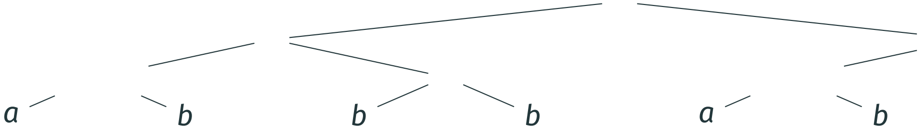
- Build a **balanced binary tree** on the input string $w = abbbab$

A general-purpose $O(\log n)$ algorithm

Fix the language $L = (ab)^*$:

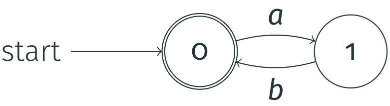


- Build a **balanced binary tree** on the input string $w = abbbab$

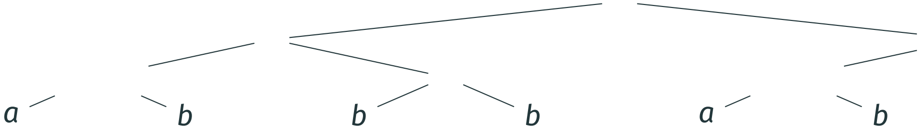


A general-purpose $O(\log n)$ algorithm

Fix the language $L = (ab)^*$:

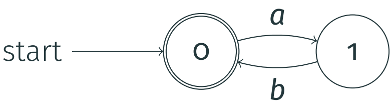


- Build a balanced binary tree on the input string $w = abbbab$
- Label each node n by the transition monoid element: all pairs $q \rightsquigarrow q'$ such that we can go from q to q' by reading the factor below n

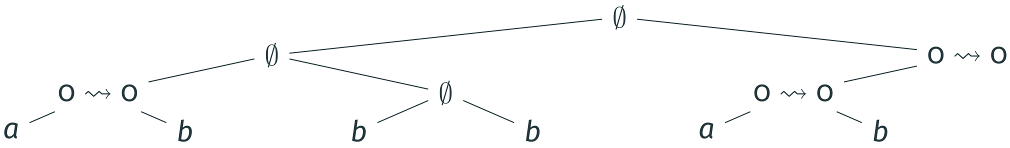


A general-purpose $O(\log n)$ algorithm

Fix the language $L = (ab)^*$:

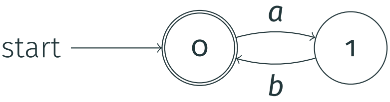


- Build a **balanced binary tree** on the input string $w = abbbab$
- Label each node n by the **transition monoid** element: all pairs $q \rightsquigarrow q'$ such that we can go from q to q' by reading the factor below n

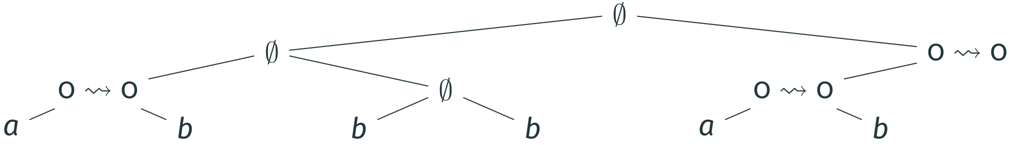


A general-purpose $O(\log n)$ algorithm

Fix the language $L = (ab)^*$:



- Build a **balanced binary tree** on the input string $w = abbbab$
- Label each node n by the **transition monoid** element: all pairs $q \rightsquigarrow q'$ such that we can go from q to q' by reading the factor below n



- The **tree root** describes if $w \in L$
- We can update the tree for each substitution **in $O(\log n)$**
- Can be improved to **$O(\log n / \log \log n)$** with a log-ary tree

Improving on $O(\log n)$ for some languages

For our language $L = (ab)^*$ we can handle updates in $O(1)$:

Improving on $O(\log n)$ for some languages

For our language $L = (ab)^*$ we can handle updates in $O(1)$:

- Check that n is even
- Count violations: a's at even positions and b's at odd positions
- Maintain this counter in constant time
- We have $w \in L$ iff there are no violations

Improving on $O(\log n)$ for some languages

For our language $L = (ab)^*$ we can handle updates in $O(1)$:

- Check that n is even
- Count violations: a's at even positions and b's at odd positions
- Maintain this counter in constant time
- We have $w \in L$ iff there are no violations

Question: what is the complexity of dynamic membership, depending on the fixed regular language L?

Summary of our results

QLZG: in $O(1)$

QSG: in $O(\log \log n)$
not in $O(1)$?

All: in $\Theta(\log n / \log \log n)$

- We identify a class **QLZG** of regular languages:
  - for any language in **QLZG**, dynamic membership is in $O(1)$
  - for any language not in **QLZG**, we can reduce from a problem that we conjecture is not in $O(1)$

Summary of our results

QLZG: in $O(1)$

QSG: in $O(\log \log n)$
not in $O(1)$?

All: in $\Theta(\log n / \log \log n)$

- We identify a class **QLZG** of regular languages:
  - for any language in **QLZG**, dynamic membership is in $O(1)$
  - for any language not in **QLZG**, we can reduce from a problem that we conjecture is not in $O(1)$
- We identify a class **QSG** of regular languages:
  - for any language in **QSG**, the problem is in $O(\log \log n)$
  - for any language not in **QSG**, it is in $\Omega(\log n / \log \log n)$ (lower bound of Skovbjerg Frandsen et al.)

Summary of our results

QLZG: in $O(1)$

QSG: in $O(\log\log n)$
not in $O(1)$?

All: in $\Theta(\log n / \log\log n)$

- We identify a class **QLZG** of regular languages:
  - for any language in **QLZG**, dynamic membership is in $O(1)$
  - for any language not in **QLZG**, we can reduce from a problem that we conjecture is not in $O(1)$
- We identify a class **QSG** of regular languages:
  - for any language in **QSG**, the problem is in $O(\log\log n)$
  - for any language not in **QSG**, it is in $\Omega(\log n / \log\log n)$ (lower bound of Skovbjerg Frandsen et al.)
- The problem is always in $O(\log n / \log\log n)$

Regular languages
and more expressive updates

Endpoint updates

- Simplest updates that change the string length: endpoint updates

Endpoint updates

- Simplest updates that change the string length: endpoint updates
  - Insert a letter at the beginning of the string, or delete the first letter
  - Insert a letter at the end of the string, or delete the last letter

Endpoint updates

- Simplest updates that change the string length: endpoint updates
  - Insert a letter at the beginning of the string, or delete the first letter
  - Insert a letter at the end of the string, or delete the last letter

$\rightarrow$ Similar to a doubly-ended queue (deque)

$\rightarrow$ Special case: sliding window

Endpoint updates

- Simplest updates that change the string length: endpoint updates
  - Insert a letter at the beginning of the string, or delete the first letter
  - Insert a letter at the end of the string, or delete the last letter
  - → Similar to a doubly-ended queue (deque)
  - → Special case: sliding window

Theorem

Dynamic membership to any fixed regular language under endpoint updates at the end of the string is possible in constant time

Proof:

Endpoint updates

- Simplest updates that change the string length: **endpoint updates**
  - Insert a letter at the **beginning** of the string, or delete the first letter
  - Insert a letter at the **end** of the string, or delete the last letter

  → Similar to a **doubly-ended queue** (deque)

  → Special case: **sliding window**

Theorem

Dynamic membership to any fixed ***regular language*** *under endpoint updates at the* ***end*** *of the string is possible in* ***constant time***

Proof: simply extend/truncate the run of a deterministic automaton

Endpoint updates

- Simplest updates that change the string length: endpoint updates
  - Insert a letter at the beginning of the string, or delete the first letter
  - Insert a letter at the end of the string, or delete the last letter
  - $\rightarrow$ Similar to a doubly-ended queue (deque)
  - $\rightarrow$ Special case: sliding window

Theorem

Dynamic membership to any fixed regular language under endpoint updates at the end of the string is possible in constant time

Proof: simply extend/truncate the run of a deterministic automaton

Theorem

The same holds for udpates at the beginning of the string

Proof:

Endpoint updates

- Simplest updates that change the string length: endpoint updates
  - Insert a letter at the beginning of the string, or delete the first letter
  - Insert a letter at the end of the string, or delete the last letter
  - $\rightarrow$ Similar to a doubly-ended queue (deque)
  - $\rightarrow$ Special case: sliding window

Theorem

Dynamic membership to any fixed regular language under endpoint updates at the end of the string is possible in constant time

Proof: simply extend/truncate the run of a deterministic automaton

Theorem

The same holds for udpates at the beginning of the string

Proof: regular languages are closed under reversal

Tractability under endpoint updates

Theorem (Louis Jachiet, CStheory (TCS.SE), 2020)

Dynamic membership to any fixed regular language under endpoint updates is possible in constant time

Tractability under endpoint updates

Theorem (Louis Jachiet, CStheory (TCS.SE), 2020)
Dynamic membership to any fixed regular language under endpoint updates is possible in constant time

Proof ("guardian algorithm"):

- Store the string in an amortized circular buffer
- We will again store the transition monoid element achieved by some factors

Tractability under endpoint updates

Theorem (Louis Jachiet, CStheory (TCS.SE), 2020)

Dynamic membership to any fixed regular language under endpoint updates is possible in constant time

Proof ("guardian algorithm"):

- Store the string in an amortized circular buffer
- We will again store the transition monoid element achieved by some factors
- Naive idea: split the string in two (put a guardian in the middle):
  - store the transition monoid elements of all suffixes of the first half
  - and of all prefixes of the second half

Tractability under endpoint updates

Theorem (Louis Jachiet, CStheory (TCS.SE), 2020)
Dynamic membership to any fixed regular language under endpoint updates is possible in constant time

Proof ("guardian algorithm"):

- Store the string in an amortized circular buffer
- We will again store the transition monoid element achieved by some factors
- Naive idea: split the string in two (put a guardian in the middle):
  - store the transition monoid elements of all suffixes of the first half
  - and of all prefixes of the second half
- Whenever the updates shift the string too much and the guardian is far from the current middle, create a new guardian at the new middle

Richer updates

Which **other updates** do we want on strings in practice?

Richer updates

Which other updates do we want on strings in practice?

- Insertion and deletion at arbitrary positions

Richer updates

Which other updates do we want on strings in practice?

- Insertion and deletion at arbitrary positions
- Split a string in two, and join two strings
  - Special case: cut and paste a factor to a different place

Richer updates

Which other updates do we want on strings in practice?

- Insertion and deletion at arbitrary positions
- Split a string in two, and join two strings
    - Special case: cut and paste a factor to a different place

Theorem (Folklore?)

Dynamic membership to any fixed regular language under insertion, substitution, deletion, split, join is possible in $O(\log n)$ time

Proof: use balancing binary trees (AVL trees) instead of the fixed complete binary tree of the $O(\log n)$ algorithm for substitutions

Doing better than $O(\log n)$ with insertions and deletions?

- For substitutions, we could do better than $O(\log n)$ for some subclasses of the regular languages
- Is the same true when allowing arbitrary insertions and deletions?

Doing better than $O(\log n)$ with insertions and deletions?

- For substitutions, we could do better than $O(\log n)$ for some subclasses of the regular languages
- Is the same true when allowing arbitrary insertions and deletions?

$\rightarrow$ No!

Theorem (Question by Louis Jachiet, result by Kasper Green Larsen, mentioned by David Eppstein, CStheory (TCS.SE), 2020)

Doing better than $O(\log n)$ with insertions and deletions?

- For substitutions, we could do **better than $O(\log n)$** for some subclasses of the regular languages
- Is the same true when allowing arbitrary insertions and deletions?

$\rightarrow$ **No!**

Theorem (Question by Louis Jachiet, result by Kasper Green Larsen, mentioned by David Eppstein, CStheory (TCS.SE), 2020)

*Maintaining membership to the language $\Sigma^* a \Sigma^*$ ("does the string contain an a") under **insertions and deletions** is in $\Omega(\log n / \log \log n)$*

Summary for dynamic membership to fixed regular languages

- With endpoint updates:

Summary for dynamic membership to fixed regular languages

- With endpoint updates: always doable in $O(1)$

Summary for dynamic membership to fixed regular languages

- With endpoint updates: always doable in $O(1)$
- With substitution updates:

Summary for dynamic membership to fixed regular languages

- With endpoint updates: always doable in $O(1)$
- With substitution updates:
  - General bound $\Theta(\log n / \log \log n)$
  - Characterization of some (all?) $O(1)$ cases and $O(\log \log n)$ cases

  $\rightarrow$ Open question: are there other classes?

Summary for dynamic membership to fixed regular languages

- With endpoint updates: always doable in $O(1)$
- With substitution updates:
  - General bound $\Theta(\log n / \log \log n)$
  - Characterization of some (all?) $O(1)$ cases and $O(\log \log n)$ cases
  - → Open question: are there other classes?
- With insertion and deletions:

Summary for dynamic membership to fixed regular languages

- With endpoint updates: always doable in $O(1)$
- With substitution updates:
  - General bound $\Theta(\log n / \log \log n)$
  - Characterization of some (all?) $O(1)$ cases and $O(\log \log n)$ cases
  - → Open question: are there other classes?
- With insertion and deletions:
  - General $O(\log n)$ bound with AVL-trees, event with split and join
  - Lower bound $\Omega(\log n / \log \log n)$ for essentially all languages

Summary for dynamic membership to fixed regular languages

- With endpoint updates: always doable in $O(1)$
- With substitution updates:
  - General bound $\Theta(\log n / \log \log n)$
  - Characterization of some (all?) $O(1)$ cases and $O(\log \log n)$ cases
  - → Open question: are there other classes?
- With insertion and deletions:
  - General $O(\log n)$ bound with AVL-trees, event with split and join
  - Lower bound $\Omega(\log n / \log \log n)$ for essentially all languages

→ Open question: combination of substitutions + endpoint updates

→ Open question: different models, e.g., doubly linked lists?

Incremental maintenance for enumeration structures

Beyond dynamic membership

- So far, we have only talked about maintaining Boolean information

  $\rightarrow$ "does the string contain a factor ab^*c?"

Beyond dynamic membership

- So far, we have only talked about maintaining Boolean information
  - $\rightarrow$ "does the string contain a factor ab^*c?"
- More interesting: maintain non-Boolean information, i.e., a set of results:
  - $\rightarrow$ "what are the factors ab^*c?"

Beyond dynamic membership

- So far, we have only talked about maintaining Boolean information
  - → "does the string contain a factor ab^*c?"

- More interesting: maintain non-Boolean information, i.e., a set of results:
  - → "what are the factors ab^*c?"

- Problem: there can be many results, so we cannot maintain the full set

Beyond dynamic membership

- So far, we have only talked about maintaining Boolean information
  $\rightarrow$ "does the string contain a factor ab^*c?"

- More interesting: maintain non-Boolean information, i.e., a set of results:
  $\rightarrow$ "what are the factors ab^*c?"

- Problem: there can be many results, so we cannot maintain the full set

- Ideas:

Beyond dynamic membership

- So far, we have only talked about maintaining **Boolean information**
  $\rightarrow$ "does the string contain a factor ab^*c?"

- More interesting: maintain non-Boolean information, i.e., a **set of results**:
  $\rightarrow$ "what are the factors ab^*c?"

- Problem: there can be **many results**, so we cannot maintain the full set

- **Ideas:**
  - "what is the **first** factor ab^*c?"

Beyond dynamic membership

- So far, we have only talked about maintaining **Boolean information**
  - $\rightarrow$ "does the string contain a factor ab^*c?"

- More interesting: maintain non-Boolean information, i.e., a **set of results**:
  - $\rightarrow$ "what are the factors ab^*c?"

- Problem: there can be **many results**, so we cannot maintain the full set

- **Ideas:**
  - "what is the **first** factor ab^*c?"
  - "**how many** factors ab^*c are there?"

Beyond dynamic membership

- So far, we have only talked about maintaining **Boolean information**
  - → "does the string contain a factor ab^*c?"
- More interesting: maintain non-Boolean information, i.e., a **set of results**:
  - → "what are the factors ab^*c?"
- Problem: there can be **many results**, so we cannot maintain the full set
- **Ideas:**
  - "what is the **first** factor ab^*c?"
  - "**how many** factors ab^*c are there?"
  - "compute an index to **test efficiently** if a factor is of the form ab^*c?"

Beyond dynamic membership

- So far, we have only talked about maintaining **Boolean information**
  - → "does the string contain a factor ab^*c?"
- More interesting: maintain non-Boolean information, i.e., a **set of results**:
  - → "what are the factors ab^*c?"
- Problem: there can be **many results**, so we cannot maintain the full set
- **Ideas:**
  - "what is the **first** factor ab^*c?"
  - "**how many** factors ab^*c are there?"
  - "compute an index to **test efficiently** if a factor is of the form ab^*c?"
  - → "compute an index to **enumerate** efficiently the factors ab^*c"

Generalizing factors

What is the right notion of **result** that we want to find in a string?

- Factors? suffixes? prefixes?

Generalizing factors

What is the right notion of **result** that we want to find in a string?

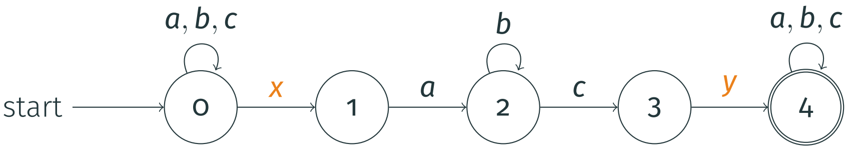
- Factors? suffixes? prefixes?
- Pairs of factors? Tuples of factors?

Generalizing factors

What is the right notion of **result** that we want to find in a string?

- Factors? suffixes? prefixes?
- Pairs of factors? Tuples of factors?

A robust notion: **automata with captures**

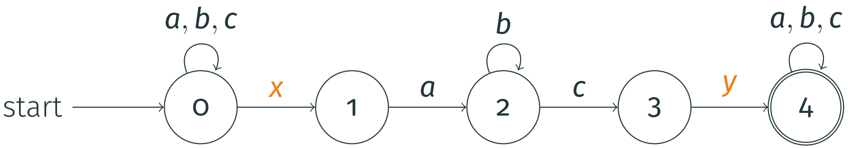


Generalizing factors

What is the right notion of result that we want to find in a string?

- Factors? suffixes? prefixes?
- Pairs of factors? Tuples of factors?

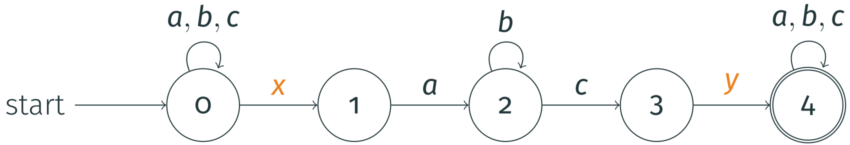
A robust notion: automata with captures



- Equivalently: monadic second-order queries with free variables
- Special case: document spanners studied in information extraction

Semantics of an automaton with captures

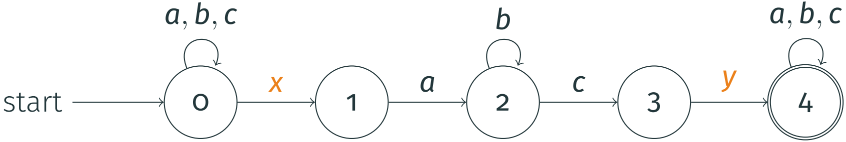
Consider the automaton with captures A on an input string w:



	0	1	2	3	4	5	6	7	8	9
w =	b	a	c	b	a	b	b	c	c	

Semantics of an automaton with captures

Consider the automaton with captures A on an input string w:

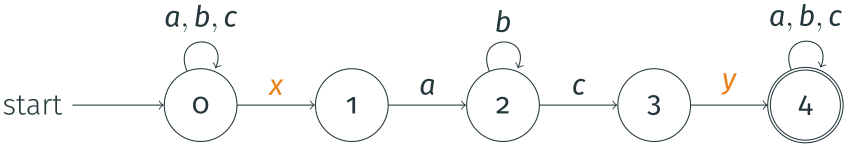


		0	1	2	3	4	5	6	7	8	9
w	$=$	b	a	c	b	a	b	b	c	c	

Set of results of A on w:

Semantics of an automaton with captures

Consider the automaton with captures A on an input string w:



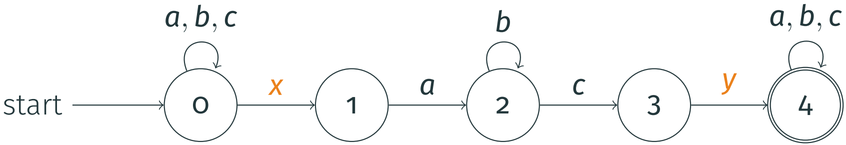
0 1 2 3 4 5 6 7 8 9

w = b a c b a b b c c

Set of results of A on w: positions where to insert x and y in w such that A accepts

Semantics of an automaton with captures

Consider the automaton with captures A on an input string w:



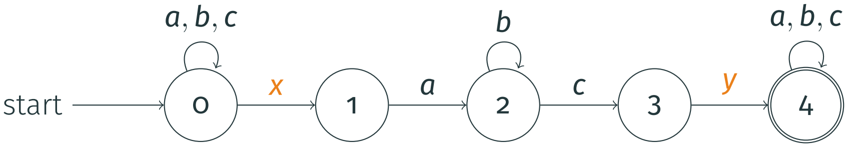
	0	1	2	3	4	5	6	7	8	9
$w =$	b	a	c	b	a	b	b	c	c	

Set of results of A on w: positions where to insert x and y in w such that A accepts

Here, two results:

Semantics of an automaton with captures

Consider the automaton with captures A on an input string w:



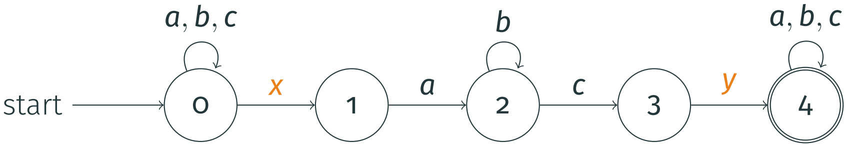
	0	1	2	3	4	5	6	7	8	9
w =	b	xa	c	yb	a	b	b	c	c	

Set of **results** of A on w: **positions** where to insert x and y in w such that A accepts

Here, two results:

Semantics of an automaton with captures

Consider the automaton with captures A on an input string w:



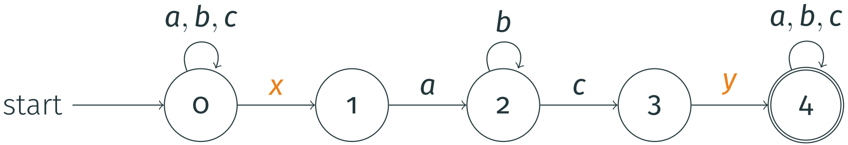
	0	1	2	3	4	5	6	7	8	9
w =	b	xa	c	yb	a	b	b	c	c	

Set of **results** of A on w: **positions** where to insert x and y in w such that A accepts

Here, two results: $\{x : 1, y : 3\}$ and

Semantics of an automaton with captures

Consider the automaton with captures A on an input string w:



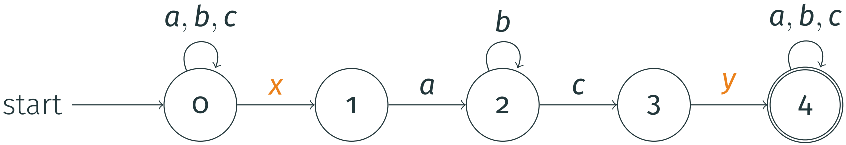
$$
\begin{array}{cccccccccccc}
 & & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
w & = & \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{b} & x\mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{c} & y\mathtt{c} &
\end{array}
$$

Set of **results** of A on w: **positions** where to insert x and y in w such that A accepts

Here, two results: $\{x : 1, y : 3\}$ and

Semantics of an automaton with captures

Consider the automaton with captures A on an input string w:



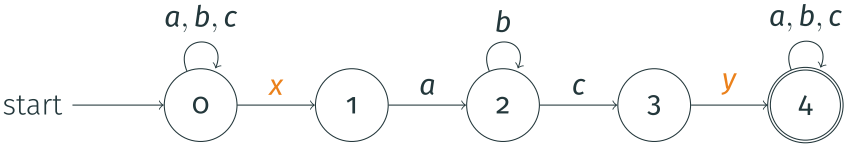
	0	1	2	3	4	5	6	7	8	9
$w =$	b	a	c	b	xa	b	b	c	yc	

Set of **results** of A on w: **positions** where to insert x and y in w such that A accepts

Here, two results: $\{x : 1, y : 3\}$ and $\{x : 4, y : 8\}$

Semantics of an automaton with captures

Consider the automaton with captures A on an input string w:



	0	1	2	3	4	5	6	7	8	9
$w =$	b	a	c	b	a	b	b	c	c	

Set of **results** of A on w: **positions** where to insert x and y in w such that A accepts

Here, two results: $\{x : 1, y : 3\}$ and $\{x : 4, y : 8\}$

In this case: endpoints of the factors which are in language ab^*c

Enumeration algorithms

We want all the results of an automaton with captures on a string

Enumeration algorithms

We want an index of all the results of an automaton with captures on a string:

Enumeration algorithms

We want an index of all the results of an automaton with captures on a string:

- Enumeration algorithm: produce the results in streaming, one after the other, without repetitions

Enumeration algorithms

We want an index of all the results of an automaton with captures on a string:

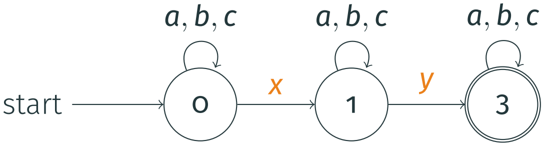
- Enumeration algorithm: produce the results in streaming, one after the other, without repetitions
- Performance: maximal delay between two consecutive results

Enumeration algorithms

We want an index of all the results of an automaton with captures on a string:

- Enumeration algorithm: produce the results in streaming, one after the other, without repetitions
- Performance: maximal delay between two consecutive results

Example: enumerate the results of



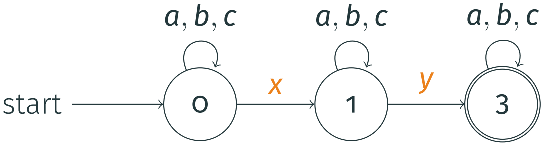
Goal: constant-delay, independent from the string length. Several uses:

Enumeration algorithms

We want an index of all the results of an automaton with captures on a string:

- Enumeration algorithm: produce the results in streaming, one after the other, without repetitions
- Performance: maximal delay between two consecutive results

Example: enumerate the results of



Goal: constant-delay, independent from the string length. Several uses:

- We can check if there is at least one result, in constant time
- We can produce all results in output-linear time

Enumeration without updates

How can we enumerate the results of an automaton with captures on a string (without updates)?

Theorem ([Florenzano et al., 2018])

For a fixed automaton with captures A, given a string w, we can prepare in $O(w)$ a data structure to enumerate the results with constant-delay

Enumeration without updates

How can we enumerate the results of an automaton with captures on a string (without updates)?

Theorem ([Florenzano et al., 2018])

For a fixed automaton with captures A, given a string w, we can prepare in $O(w)$ a data structure to enumerate the results with constant-delay

Proof:

- Do a product of A and w

Enumeration without updates

How can we enumerate the results of an automaton with captures on a string (without updates)?

Theorem ([Florenzano et al., 2018])

For a fixed automaton with captures A, given a string w, we can prepare in $O(w)$ a data structure to enumerate the results with constant-delay

Proof:

- Do a product of A and w
- Annotate variable transitions with the position in w

Enumeration without updates

How can we enumerate the results of an **automaton with captures** on a string (without updates)?

Theorem ([Florenzano et al., 2018])

*For a fixed **automaton with captures** A, given a string w, we can prepare in $O(w)$ a data structure to enumerate the results with **constant-delay***

Proof:

- Do a **product** of A and w
- Annotate **variable transitions** with the position in w
- Replace **non-variable transitions** by ϵ

Enumeration without updates

How can we enumerate the results of an **automaton with captures** on a string (without updates)?

Theorem ([Florenzano et al., 2018])

For a fixed ***automaton with captures*** *A, given a string w, we can prepare in $O(w)$ a data structure to enumerate the results with* ***constant-delay***

Proof:

- Do a **product** of A and w
- Annotate **variable transitions** with the position in w
- Replace **non-variable transitions** by ϵ
- Do a form of **ϵ-removal** (can be done in linear time here)

Enumeration without updates

How can we enumerate the results of an automaton with captures on a string (without updates)?

Theorem ([Florenzano et al., 2018])

For a fixed automaton with captures A, given a string w, we can prepare in $O(w)$ a data structure to enumerate the results with constant-delay

Proof:

- Do a product of A and w
- Annotate variable transitions with the position in w
- Replace non-variable transitions by ϵ
- Do a form of ϵ-removal (can be done in linear time here)
- Enumerate the paths of the resulting DAG

Enumeration without updates

How can we enumerate the results of an automaton with captures on a string (without updates)?

Theorem ([Florenzano et al., 2018])

For a fixed automaton with captures A, given a string w, we can prepare in $O(w)$ a data structure to enumerate the results with constant-delay

Proof:

- Do a product of A and w
- Annotate variable transitions with the position in w
- Replace non-variable transitions by ϵ
- Do a form of ϵ-removal (can be done in linear time here)
- Enumerate the paths of the resulting DAG

$\rightarrow$ Can we incrementally maintain enumeration structures under updates?

Maintaining an enumeration structure

Theorem ([Niewerth and Segoufin, 2018])

We can maintain a constant-delay enumeration structure for automata with captures under insertion, substitution, and deletion updates in time $O(\log n)$

Proof: complex formal language results (Krohn-Rhodes theory).

Maintaining an enumeration structure

Theorem ([Niewerth and Segoufin, 2018])

We can maintain a constant-delay enumeration structure for automata with captures under insertion, substitution, and deletion updates in time $O(\log n)$

Proof: complex formal language results (Krohn-Rhodes theory).

Theorem ([Schmid and Schweikardt, 2022])

The same holds with join and split (and more complex edit operations) but with logarithmic delay.

Proof: balancing straight-line programs (SLP)

Maintaining an enumeration structure

Theorem ([Niewerth and Segoufin, 2018])

*We can maintain a **constant-delay** enumeration structure for automata with captures under **insertion, substitution, and deletion updates** in time $O(\log n)$*

Proof: complex formal language results (Krohn-Rhodes theory).

Theorem ([Schmid and Schweikardt, 2022])

*The same holds with **join** and **split** (and more complex edit operations) but with **logarithmic delay**.*

Proof: balancing straight-line programs (SLP)

Conjecture

*Both are doable: support **join** and **split** in time $O(\log n)$ and **constant-delay***

Also: support **endpoint updates** with constant time and constant-delay

Improving the complexity

- Can we have a complexity **better than $O(\log n)$?**

Improving the complexity

- Can we have a complexity **better than $O(\log n)$?**
- Idea: restricting to **specific languages** of automata with captures (like in our classification of regular languages under updates)

Improving the complexity

- Can we have a complexity **better than $O(\log n)$?**
- Idea: restricting to **specific languages** of automata with captures (like in our classification of regular languages under updates)

$\rightarrow$ **Open research question!**

Conclusion and perspectives

High-level summary

- We want to **incrementally maintain** information on a string under updates
- Simple Boolean problem: **dynamic membership** to a regular language
- More expressive problem: **maintaining an enumeration structure** for an automaton with captures
- **General case:** everything should always be in $O(\log n)$ (?)
- Better cases:
  - **Endpoint updates:** everything is in $O(1)$ (?)
  - **Substitution updates** for **dynamic membership**: $O(1)$ or $O(\log \log n)$ or $\Theta(\log n / \log \log n)$ (... or?) depending on the language
- **Future research:** identify more cases below $O(\log n)$

Future directions

- Maintaining a structure for **infix testing**, membership testing, etc.
  - → Without updates: factorization forests, or structure of [Bojańczyk, 2009]
  - → With **substitutions**: amounts to **incremental maintenance** for another language
  - → With **endpoint updates**: should be possible in constant-time too

Future directions

- Maintaining a structure for **infix testing**, membership testing, etc.
  - → Without updates: factorization forests, or structure of [Bojańczyk, 2009]
  - → With **substitutions**: amounts to **incremental maintenance** for another language
  - → With **endpoint updates**: should be possible in constant-time too
- Maintaining a **count**: number of results, acceptance probability, etc.

Future directions

- Maintaining a structure for **infix testing**, membership testing, etc.
  - → Without updates: factorization forests, or structure of [Bojańczyk, 2009]
  - → With **substitutions**: amounts to **incremental maintenance** for another language
  - → With **endpoint updates**: should be possible in constant-time too
- Maintaining a **count**: number of results, acceptance probability, etc.
- Extending from **regular languages** to **context-free languages**
  - → Related work: **incremental parsing**?
  - → Data structures for **enumeration**: [Peterfreund, 2021] [Amarilli et al., 2022]
  - → More research and more algebraic tools needed

Future directions

- Maintaining a structure for **infix testing**, membership testing, etc.
  - → Without updates: factorization forests, or structure of [Bojańczyk, 2009]
  - → With **substitutions**: amounts to **incremental maintenance** for another language
  - → With **endpoint updates**: should be possible in constant-time too
- Maintaining a **count**: number of results, acceptance probability, etc.
- Extending from **regular languages** to **context-free languages**
  - → Related work: **incremental parsing**?
  - → Data structures for **enumeration**: [Peterfreund, 2021] [Amarilli et al., 2022]
  - → More research and more algebraic tools needed
- Extending from **string** to **trees**
  - → Doable in $O(\log^2 n)$ [Losemann and Martens, 2014]
  - → **Still $O(\log n)$?** [Amarilli et al., 2019], proof currently broken
  - → Better than $O(\log n)$: more research and more algebraic tools needed

Future directions

- Maintaining a structure for **infix testing**, membership testing, etc.
  - $\rightarrow$ Without updates: factorization forests, or structure of [Bojańczyk, 2009]
  - $\rightarrow$ With **substitutions**: amounts to **incremental maintenance** for another language
  - $\rightarrow$ With **endpoint updates**: should be possible in constant-time too
- Maintaining a **count**: number of results, acceptance probability, etc.
- Extending from **regular languages** to **context-free languages**
  - $\rightarrow$ Related work: **incremental parsing**?
  - $\rightarrow$ Data structures for **enumeration**: [Peterfreund, 2021] [Amarilli et al., 2022]
  - $\rightarrow$ More research and more algebraic tools needed
- Extending from **string** to **trees**
  - $\rightarrow$ Doable in $O(\log^2 n)$ [Losemann and Martens, 2014]
  - $\rightarrow$ **Still $O(\log n)$?** [Amarilli et al., 2019], proof currently broken
  - $\rightarrow$ Better than $O(\log n)$: more research and more algebraic tools needed

Thanks for your attention!

References i

Amarilli, A., Bourhis, P., Mengel, S., and Niewerth, M. (2019).
Enumeration on Trees With Tractable Combined Complexity and Efficient Updates.
In *PODS*.

Amarilli, A., Jachiet, L., Muñoz, M., and Riveros, C. (2022).
Efficient Enumeration Algorithms for Annotated Grammars.
In *PODS*.

Amarilli, A., Jachiet, L., and Paperman, C. (2021).
Dynamic Membership for Regular Languages.
In *ICALP*.

Amarilli, A. and Paperman, C. (2021).
Locality and Centrality: The Variety ZG.
Under review.

Bojańczyk, M. (2009).
Factorization Forests.
In *DLT*.

Eppstein, D.
On the Complexity of a "List" Datastructure in the RAM Model.
Theoretical Computer Science Stack Exchange.
URL: `https://cstheory.stackexchange.com/q/46749` (version: 2020-05-25).

Florenzano, F., Riveros, C., Ugarte, M., Vansummeren, S., and Vrgoc, D. (2018).
Constant Delay Algorithms for Regular Document Spanners.
In *PODS*.

Jachiet, L.
Constraints on Sliding Windows.
Theoretical Computer Science Stack Exchange.
URL: `https://cstheory.stackexchange.com/q/46762` (version: 2020-05-06).

Losemann, K. and Martens, W. (2014).
MSO Queries on Trees: Enumerating Answers Under Updates.
In *CSL-LICS*.

References iv

Niewerth, M. and Segoufin, L. (2018).
Enumeration of MSO Queries on Strings with Constant Delay and Logarithmic Updates.
In *PODS*.

Peterfreund, L. (2021).
Grammars for Document Spanners.
In *ICDT*.

Schmid, M. and Schweikardt, N. (2022).
Query Evaluation Over SLP-Represented Document Databases With Complex Document Editing.
In *PODS*.

Skovbjerg Frandsen, G., Miltersen, P. B., and Skyum, S. (1997).
Dynamic Word Problems.
JACM, 44(2).

Other research themes

- 00
- 01
- 10

⋮

- Enumeration algorithms, links to circuit classes
  - Enumeration for regular spanners and grammars
  - In-order enumeration
  - Connections to knowledge compilation

Other research themes

- 00
- 01
- 10

⋮

- Enumeration algorithms, links to circuit classes
  - Enumeration for regular spanners and grammars
  - In-order enumeration
  - Connections to knowledge compilation

1

0 ~~0~~ 0 1

- Efficient maintenance of query results on dynamic data
  - Supporting membership queries, counts, enumeration structures...
  - For regular languages, regular tree languages, context-free languages...
  - On string, trees, graphs...
  - Under substitution updates or other updates

Other research themes

- 00
- 01
- 10

⋮

- Enumeration algorithms, links to circuit classes
  - Enumeration for regular spanners and grammars
  - In-order enumeration
  - Connections to knowledge compilation

1

0 ~~0~~ 0 1

- Efficient maintenance of query results on dynamic data
  - Supporting membership queries, counts, enumeration structures...
  - For regular languages, regular tree languages, context-free languages...
  - On string, trees, graphs...
  - Under substitution updates or other updates

0? 50%

1? 50%

- Query evaluation on probabilistic data
  - Dichotomies for homomorphism-closed queries
  - Uniform model counting
  - Treewidth-based and grid-minor-based methods

Other research themes

- 00
- 01
- 10

⋮

- **Enumeration algorithms**, links to **circuit classes**
  - Enumeration for **regular spanners** and **grammars**
  - **In-order** enumeration
  - Connections to **knowledge compilation**

1

0 ~~0~~ 0 1

- **Efficient maintenance** of query results on **dynamic data**
  - Supporting **membership queries**, counts, enumeration structures...
  - For **regular languages**, regular tree languages, context-free languages...
  - On **string**, trees, graphs...
  - Under **substitution updates** or other updates

0? 50%

1? 50%

- **Query evaluation** on **probabilistic data**
  - Dichotomies for **homomorphism-closed queries**
  - **Uniform** model counting
  - **Treewidth-based** and grid-minor-based methods
- **Database theory**, provenance, logics...

Advertisement: TCS4F and “No free view? No review!”

TheoretiCS

A new open-access journal for theoretical computer science
(managing editor with Nathanaël Fijalkow)

Advertisement: TCS4F and “No free view? No review!”

TheoretiCS

A new open-access journal for theoretical computer science
(managing editor with Nathanaël Fijalkow)

A pledge to reduce the carbon footprint of your research travels

`www.tcs4f.org`

(with Thomas Schwentick, Thomas Colcombet, Hugo Férée)

TCS4F
THEORETICAL
COMPUTER
SCIENTISTS
FOR FUTURE

Advertisement: TCS4F and “No free view? No review!”

TheoretiCS

A new **open-access journal** for **theoretical computer science** (managing editor with Nathanaël Fijalkow)

A pledge to **reduce the carbon footprint** of your research travels

`www.tcs4f.org`

(with Thomas Schwentick, Thomas Colcombet, Hugo Férée)

TCS4F
THEORETICAL COMPUTER SCIENTISTS FOR FUTURE

NO FREE VIEW?
NO REVIEW!

A pledge **not to review** for conferences and journals that do not publish their research as **open access**

`www.nofreeviewnoreview.org`

(with Antonin Delpeuch)