Regular Languages: Some New Problems and Algorithms

Antoine Amarilli

November 26, 2025

Joint work with: Corentin Barloy, Pierre Bourhis, İsmail İlkan Ceylan, Sven Dziadek, Octave Gaspard, Wolfgang Gatterbauer, Paweł Gawrychowski, Benoît Groz, Santiago Guzman Pro, Louis Jachiet, Sébastien Labbé, Neha Makhija, Kuldeep Meel, Stefan Mengel, Mikaël Monet, Martín Muñoz, Matthias Niewerth, Charles Paperman, Paul Raphaël, Tina Ringleb, Cristian Riveros, Sylvain Salvati, Luc Segoufin, Tim Van Bremen, Nicole Wein

Who am I?

Antoine Amarilli, `a3nm.net`

Researcher (Advanced Research Position) at Inria Lille

On leave from Télécom Paris (associate professor)

Who am I?

Antoine Amarilli, `a3nm.net`

Researcher (Advanced Research Position) at Inria Lille

On leave from Télécom Paris (associate professor)

- Student at École normale supérieure, Paris
  → Master parisien de recherche en informatique (MPRI)

Who am I?

Antoine Amarilli, `a3nm.net`

Researcher (Advanced Research Position) at Inria Lille

On leave from Télécom Paris (associate professor)

- Student at École normale supérieure, Paris
  → Master parisien de recherche en informatique (MPRI)

- 2013–2016: PhD at Télécom Paris

Who am I?

Antoine Amarilli, `a3nm.net`

Researcher (Advanced Research Position) at Inria Lille

On leave from Télécom Paris (associate professor)

- Student at École normale supérieure, Paris
  - → Master parisien de recherche en informatique (MPRI)
- 2013–2016: PhD at Télécom Paris
- 2016–: Associate professor at Télécom Paris

Who am I?

Antoine Amarilli, `a3nm.net`

Researcher (Advanced Research Position) at Inria Lille

On leave from Télécom Paris (associate professor)

- Student at École normale supérieure, Paris
  - → Master parisien de recherche en informatique (MPRI)
- 2013–2016: PhD at Télécom Paris
- 2016–: Associate professor at Télécom Paris
- 2024–2026(?): Researcher at Inria Lille

Which research am I doing?

- As a student: originally interested in programming (not theory!)
- Various practical internships steered me away from practice

Which research am I doing?

- As a student: originally interested in **programming** (not theory!)
- Various practical internships steered me **away from practice**
- PhD in **database theory**
  → Using theoretical methods on "practically motivated" topics

Which research am I doing?

- As a student: originally interested in **programming** (not theory!)
- Various practical internships steered me **away from practice**
- PhD in **database theory**
  - $\rightarrow$ Using theoretical methods on “practically motivated” topics
- Since then: Going more towards **pure TCS**

Which research am I doing?

- As a student: originally interested in **programming** (not theory!)
- Various practical internships steered me **away from practice**
- PhD in **database theory**
  - → Using theoretical methods on “practically motivated” topics
- Since then: Going more towards **pure TCS**

Current research:

- Areas: database theory, formal language theory, graphs, algorithms, logics

Which research am I doing?

- As a student: originally interested in programming (not theory!)
- Various practical internships steered me away from practice
- PhD in database theory
  → Using theoretical methods on “practically motivated” topics
- Since then: Going more towards pure TCS

Current research:

- Areas: database theory, formal language theory, graphs, algorithms, logics
- Central theme: query evaluation
  → determine if data satisfies a query

Which research am I doing?

- As a student: originally interested in programming (not theory!)
- Various practical internships steered me away from practice
- PhD in database theory

  $\rightarrow$ Using theoretical methods on “practically motivated” topics
- Since then: Going more towards pure TCS

Current research:

- Areas: database theory, formal language theory, graphs, algorithms, logics
- Central theme: query evaluation

  $\rightarrow$ determine if data satisfies a query
- Today’s talk: regular language membership

  $\rightarrow$ determine if a word belongs to a regular language

How am I doing research?

Some personal policies and collective initiatives to improve academic research

How am I doing research?

Some personal policies and collective initiatives to improve academic research

Climate crisis: unsustainable plane travel to CS conferences

- Since 2020: no plane travel (except 4-month sabbatical in 2023)
- Boycotting reviewing and involvement in conferences with mandatory in-person attendance
  → both an inclusivity problem and a sustainability problem
- TCS4F: measure and limit the carbon footprint of TCS research

TCS4F
THEORETICAL
COMPUTER
SCIENTISTS
FOR FUTURE

How am I doing research?

Some personal policies and collective initiatives to improve academic research

Climate crisis: unsustainable plane travel to CS conferences

- Since 2020: no plane travel (except 4-month sabbatical in 2023)
- Boycotting reviewing and involvement in conferences with mandatory in-person attendance
  - → both an inclusivity problem and a sustainability problem
- TCS4F: measure and limit the carbon footprint of TCS research

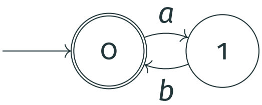
Open access: TCS research is published behind paywalls and/or with high article processing charges, because of parasitic publishers

- Since 2016: not reviewing for closed-access conferences
- NFVNR: encourage other researchers to do the same

Membership to regular languages

Regular languages

Regular languages are a robust framework for constant-memory computation:

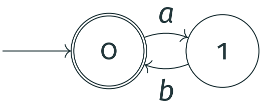


- Correspond to regular expressions, e.g., $(ab^*c|d)^*$
- Correspond to finite automata
  - Can be deterministic (DFA) or nondeterministic (NFA)

Regular languages

Regular languages are a robust framework for constant-memory computation:

- Correspond to regular expressions, e.g., $(ab^*c|d)^*$
- Correspond to finite automata
  - Can be deterministic (DFA) or nondeterministic (NFA)



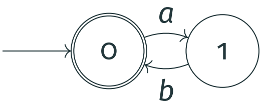
Key question: Membership problem

Given a word w and a regular language L, does $w \in L$?

Regular languages

Regular languages are a robust framework for constant-memory computation:

- Correspond to regular expressions, e.g., $(ab^*c|d)^*$
- Correspond to finite automata
    - Can be deterministic (DFA) or nondeterministic (NFA)



Key question: Membership problem

Given a word w and a regular language L, does $w \in L$?

The computational complexity can be studied in two settings:

- Data complexity: the language L is fixed and the input is w
- Combined complexity: the input is both w and some representation of L

Membership problem

What is the complexity of membership?

- In data complexity: can be decided in $O(|w|)$ on an input word w
  - $O(1)$ possible for some languages [Aaronson et al., 2019]

$\rightarrow$ We can check the word length modulo p in $O(1)$

$\rightarrow$ We can check the first and last p letters in $O(1)$

Membership problem

What is the complexity of membership?

- In data complexity: can be decided in $O(|w|)$ on an input word w
  - $O(1)$ possible for some languages [Aaronson et al., 2019]

  $\rightarrow$ We can check the word length modulo p in $O(1)$

  $\rightarrow$ We can check the first and last p letters in $O(1)$

- In combined complexity:
  - Given a DFA A, can be decided in $O(|w| + |A|)$ by running the DFA

Membership problem

What is the complexity of membership?

- In data complexity: can be decided in $O(|w|)$ on an input word w
  - $O(1)$ possible for some languages [Aaronson et al., 2019]
  - $\rightarrow$ We can check the word length modulo p in $O(1)$
  - $\rightarrow$ We can check the first and last p letters in $O(1)$
- In combined complexity:
  - Given a DFA A, can be decided in $O(|w| + |A|)$ by running the DFA
  - Given an NFA A, can be decided in $O(|w| \cdot |A|)$ by state-set simulation

Membership problem (state set simulation)

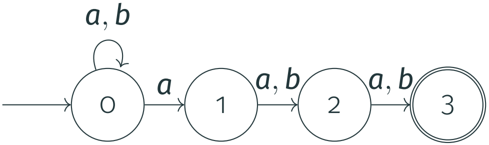
Given an NFA A and word w, check $w \in \mathrm{L}(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

Membership problem (state set simulation)

Given an NFA A and word w, check $w \in L(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

L: "the 3rd rightmost letter is an a"

$w = \; a \; b \; a \; a \; b \; b$



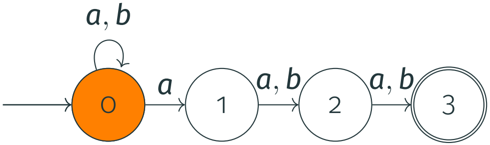
Membership problem (state set simulation)

Given an NFA A and word w, check $w \in L(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

L: "the 3rd rightmost letter is an a"

$w = \ a \ b \ a \ a \ b \ b$

$\uparrow$



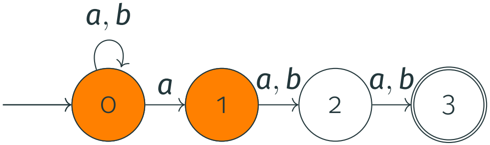
Membership problem (state set simulation)

Given an NFA A and word w, check $w \in L(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

L: “the 3rd rightmost letter is an a”

$w = \ a \ \ b \ \ a \ \ a \ \ b \ \ b$

$\quad\ \uparrow$



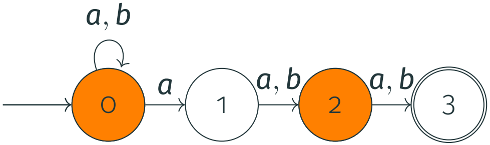
Membership problem (state set simulation)

Given an NFA A and word w, check $w \in L(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

L: "the 3rd rightmost letter is an a"

$w = \; a \; b \; a \; a \; b \; b$

$\uparrow$



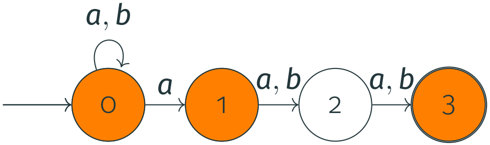
Membership problem (state set simulation)

Given an NFA A and word w, check $w \in L(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

L: "the 3rd rightmost letter is an a"

$w = a \; b \; a \; a \; b \; b$

$\uparrow$



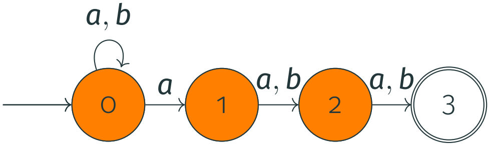
Membership problem (state set simulation)

Given an NFA A and word w, check $w \in L(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

L: "the 3rd rightmost letter is an a"

$w = a \ b \ a \ a \ b \ b$

$\uparrow$



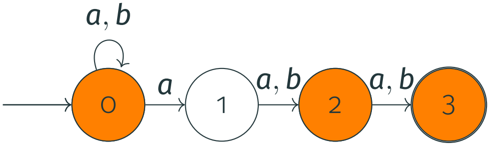
Membership problem (state set simulation)

Given an NFA A and word w, check $w \in \mathrm{L}(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

L: "the 3rd rightmost letter is an a"

$w = \; a \;\; b \;\; a \;\; a \;\; b \;\; b$

$\uparrow$



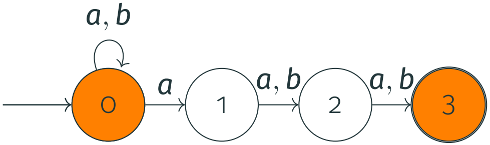
Membership problem (state set simulation)

Given an NFA A and word w, check $w \in L(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

L: "the 3rd rightmost letter is an a"

$w = a\ b\ a\ a\ b\ b$

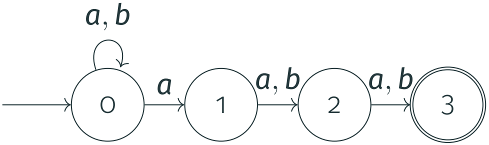
$\uparrow$



Membership problem (state set simulation)

Given an NFA A and word w, check $w \in \mathrm{L}(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

L: "the 3rd rightmost letter is an a"
$w = \; a \;\; b \;\; a \;\; a \;\; b \;\; b$

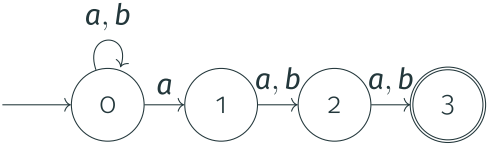


- Conditional lower bound: no general $O((|w| \cdot |A|)^{1-\epsilon})$ algorithm for any $\epsilon > 0$ assuming SETH [Backurs and Indyk, 2016]

Membership problem (state set simulation)

Given an NFA A and word w, check $w \in \mathrm{L}(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

L: "the 3rd rightmost letter is an a"
$w = \ a \ \ b \ \ a \ \ a \ \ b \ \ b$



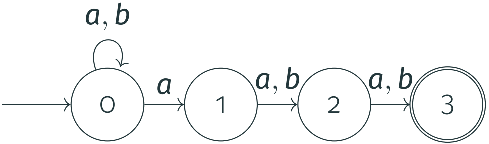
- Conditional lower bound: no general $O((|w| \cdot |A|)^{1-\epsilon})$ algorithm for any $\epsilon > 0$ assuming SETH [Backurs and Indyk, 2016]

$\rightarrow$ Better bounds possible for specific language families, e.g., subword/superword-closed languages [A., Manea, Ringleb, Schmid, 2025]

Membership problem (state set simulation)

Given an NFA A and word w, check $w \in L(A)$ in $O(|w| \cdot |A|)$ by state-set simulation

L: "the 3rd rightmost letter is an a"
$w = \ a \ b \ a \ a \ b \ b$



- Conditional lower bound: no general $O((|w| \cdot |A|)^{1-\epsilon})$ algorithm for any $\epsilon > 0$ assuming SETH [Backurs and Indyk, 2016]

$\rightarrow$ Better bounds possible for specific language families, e.g., subword/superword-closed languages [A., Manea, Ringleb, Schmid, 2025]

In this talk we will look at extensions of the membership problem

Roadmap

I will present **four extensions** of regular language membership...

- Membership for **partial words** and **probabilistic words**
- **Incremental maintenance** of membership
- **Enumeration** of word factors (beyond Boolean queries)
- Regular language problems on **graphs**

... and will sketch more directions at the end.

Partial and Probabilistic Words

Partial and probabilistic words

- Partial word: word with holes
  - → e.g., $a\_b\_$ on alphabet $\Sigma = \{a, b\}$
- Possible completions: filling the holes with letters
  - → here, 4 possible completions
- Membership for partial words to a fixed language L:
  Given a partial word w, how many completions of w belong to L?
  - → For $L = a^* b^*$, 2 of the 4 completions

Partial and probabilistic words

- Partial word: word with holes
  - $\rightarrow$ e.g., $a\sqcup b\sqcup$ on alphabet $\Sigma = \{a, b\}$
- Possible completions: filling the holes with letters
  - $\rightarrow$ here, 4 possible completions
- Membership for partial words to a fixed language L:
  Given a partial word w, how many completions of w belong to L?
  - $\rightarrow$ For $L = a^*b^*$, 2 of the 4 completions

Generalization: probabilistic words

- Each hole specifies a probability distribution over the alphabet
  - $\rightarrow$ e.g., $w = a \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix} b \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix}$
- This defines a probability distribution on possible completions

Partial and probabilistic words

- **Partial word**: word with holes
  → e.g., $a\_b\_$ on alphabet $\Sigma = \{a, b\}$
- **Possible completions:** filling the holes with letters
  → here, 4 possible completions
- **Membership for partial words** to a fixed language L:
  Given a partial word w, **how many completions** of w belong to L?
  → For $L = a^*b^*$, 2 of the 4 completions

Generalization: **probabilistic words**

- Each hole specifies a **probability distribution** over the alphabet
  → e.g., $w = a \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix} b \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix}$
- This defines a **probability distribution** on possible completions
- **Membership for probabilistic words** to L: Given a probabilistic word w, what is the **total probability** of the completions of w that belong to L?

Results on membership for probabilistic words: data complexity

Data complexity (fixed regular language L):

Results on membership for probabilistic words: data complexity

Data complexity (fixed regular language L): in linear time (up to arithmetic costs)

- Build a DFA for L (or just an unambiguous automaton or UFA)
- Do dynamic programming: read the probabilistic word w and remember the probability vector on the states

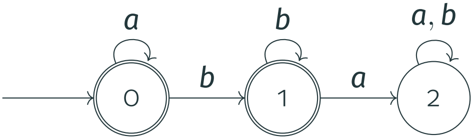
Results on membership for probabilistic words: data complexity

Data complexity (fixed regular language L): in linear time (up to arithmetic costs)

- Build a DFA for L (or just an unambiguous automaton or UFA)
- Do dynamic programming: read the probabilistic word w and remember the probability vector on the states

$L : a^* b^*$

$$w = a \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix} b \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix}$$



Results on membership for probabilistic words: data complexity

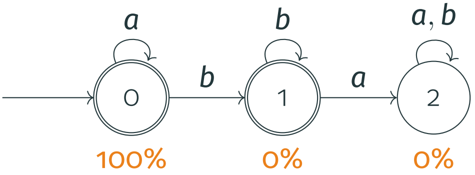
Data complexity (fixed regular language L): in linear time (up to arithmetic costs)

- Build a DFA for L (or just an unambiguous automaton or UFA)
- Do dynamic programming: read the probabilistic word w and remember the probability vector on the states

$L : a^* b^*$

$$w = a \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix} b \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix}$$

↑



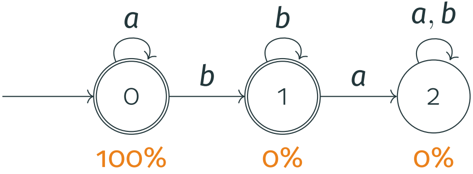
Results on membership for probabilistic words: data complexity

Data complexity (fixed regular language L): in linear time (up to arithmetic costs)

- Build a DFA for L (or just an unambiguous automaton or UFA)
- Do dynamic programming: read the probabilistic word w and remember the probability vector on the states

$L : a^* b^*$

$$w = a \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix} b \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix}$$



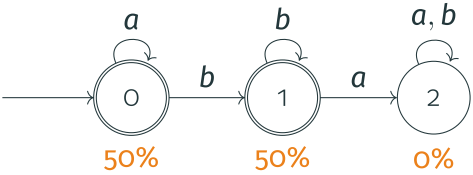
Results on membership for probabilistic words: data complexity

Data complexity (fixed regular language L): in linear time (up to arithmetic costs)

- Build a DFA for L (or just an unambiguous automaton or UFA)
- Do dynamic programming: read the probabilistic word w and remember the probability vector on the states

$L : a^* b^*$

$$w = a \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix} b \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix}$$



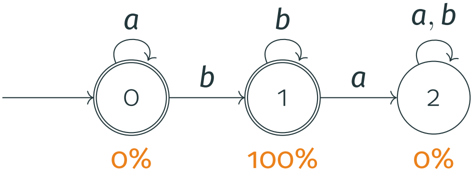
Results on membership for probabilistic words: data complexity

Data complexity (fixed regular language L): in linear time (up to arithmetic costs)

- Build a DFA for L (or just an unambiguous automaton or UFA)
- Do dynamic programming: read the probabilistic word w and remember the probability vector on the states

$L : a^* b^*$

$$w = a \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix} b \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix}$$



Results on membership for probabilistic words: data complexity

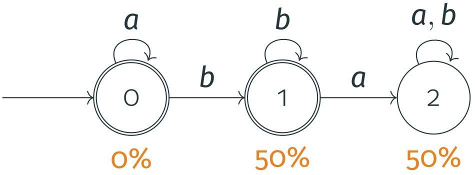
Data complexity (fixed regular language L): in linear time (up to arithmetic costs)

- Build a DFA for L (or just an unambiguous automaton or UFA)
- Do dynamic programming: read the probabilistic word w and remember the probability vector on the states

$L : a^*b^*$

$$w = a \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix} b \begin{pmatrix} a : 1/2 \\ b : 1/2 \end{pmatrix}$$

$\uparrow$



Membership for probabilistic words: Combined complexity

In combined complexity:

- It is #P-hard in general but can be approximated (FPRAS) [Arenas et al., 2021]
- It is in PTIME when the input is a DFA or UFA
- Also PTIME for k-ambiguous automata? (following [Stearns and Hunt III, 1985])

Membership for probabilistic words: Combined complexity

In combined complexity:

- It is #P-hard in general but can be approximated (FPRAS) [Arenas et al., 2021]
- It is in PTIME when the input is a DFA or UFA
- Also PTIME for k-ambiguous automata? (following [Stearns and Hunt III, 1985])

Generalizations to context-free grammars: [A., Monet, Raphaël, Salvati, 2025]

- It is in PTIME for unambiguous CFGs
- It is #P-hard already for some 2-ambiguous linear CFGs
- Other tractable cases:
  - Polyslender CFGs, e.g., $\{a^n b^n \mid n \in \mathbb{N}\}$
  - Some unambiguous counter automata
  - Some ad-hoc languages, e.g., concatenations of two palindromes

Dynamic Membership

Dynamic membership for regular languages

- Fix a regular language L
  - $\rightarrow$ E.g., $L = (ab)^*$

Dynamic membership for regular languages

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

Dynamic membership for regular languages

- Fix a regular language L
  - $\rightarrow$ E.g., $L = (ab)^*$
- Read an input word w with $n := |w|$
  - $\rightarrow$ E.g., $w = abbbab$
- Determine membership: do we have $w \in L$
  - $\rightarrow$ Here, no

Dynamic membership for regular languages

- Fix a regular language L
  - → E.g., $L = (ab)^*$
- Read an input word w with $n := |w|$
  - → E.g., $w = abbbab$
- Determine membership: do we have $w \in L$
  - → Here, no
- Efficiently maintain whether $w \in L$ under updates to w
  - → Measured in worst-case time per update operation (data complexity)

Dynamic membership for regular languages

- Fix a regular language L
  → E.g., $L = (ab)^*$
- Read an input word w with $n := |w|$
  → E.g., $w = abbbab$
- Determine membership: do we have $w \in L$
  → Here, no
- Efficiently maintain whether $w \in L$ under updates to w
  → Measured in worst-case time per update operation (data complexity)

Many different kinds of updates:

- Endpoint updates (next slide)
- Substitution updates (main focus)
- Other updates (insert/delete, cut and paste, etc.)

Dynamic membership under endpoint updates

- Push-right and pop-right: add/remove letters at the end of the word (stack)
  - We can maintain membership in...

Dynamic membership under endpoint updates

- Push-right and pop-right: add/remove letters at the end of the word (stack)
  - We can maintain membership in... $O(1)$ per update

$\rightarrow$ Simply extend / truncate the run of a DFA

Dynamic membership under endpoint updates

- Push-right and pop-right: add/remove letters at the end of the word (stack)
  - We can maintain membership in... $O(1)$ per update
  - → Simply extend / truncate the run of a DFA
- Push-right and pop-left: add at the end, remove at the beginning (queue)
  - We can maintain membership in...

Dynamic membership under endpoint updates

- Push-right and pop-right: add/remove letters at the end of the word (stack)
  - We can maintain membership in... $O(1)$ per update
  - $\rightarrow$ Simply extend / truncate the run of a DFA

- Push-right and pop-left: add at the end, remove at the beginning (queue)
  - We can maintain membership in... $O(1)$ per update
  - Not obvious (relates to simulating a queue with stacks), cf [Ganardi et al., 2022]
  - Note: the space complexity has also been characterized [Ganardi et al., 2025]

Dynamic membership under endpoint updates

- Push-right and pop-right: add/remove letters at the end of the word (stack)
    - We can maintain membership in… $O(1)$ per update

  → Simply extend / truncate the run of a DFA

- Push-right and pop-left: add at the end, remove at the beginning (queue)
    - We can maintain membership in… $O(1)$ per update
    - Not obvious (relates to simulating a queue with stacks), cf [Ganardi et al., 2022]
    - Note: the space complexity has also been characterized [Ganardi et al., 2025]

- Push/pop left/right updates (deque)
    - We can maintain membership in…

Dynamic membership under endpoint updates

- Push-right and pop-right: add/remove letters at the end of the word (stack)
  - We can maintain membership in... $O(1)$ per update

  $\rightarrow$ Simply extend / truncate the run of a DFA

- Push-right and pop-left: add at the end, remove at the beginning (queue)
  - We can maintain membership in... $O(1)$ per update
  - Not obvious (relates to simulating a queue with stacks), cf [Ganardi et al., 2022]
  - Note: the space complexity has also been characterized [Ganardi et al., 2025]

- Push/pop left/right updates (deque)
  - We can maintain membership in... $O(1)$ (above algorithm extends)

Dynamic membership under endpoint updates

- Push-right and pop-right: add/remove letters at the end of the word (stack)
  - We can maintain membership in... $O(1)$ per update
  - → Simply extend / truncate the run of a DFA
- Push-right and pop-left: add at the end, remove at the beginning (queue)
  - We can maintain membership in... $O(1)$ per update
  - Not obvious (relates to simulating a queue with stacks), cf [Ganardi et al., 2022]
  - Note: the space complexity has also been characterized [Ganardi et al., 2025]
- Push/pop left/right updates (deque)
  - We can maintain membership in... $O(1)$ (above algorithm extends)

OK, what about updates inside the word?

Dynamic membership under substitution updates

Say we have $L = (ab)^*$ and we read a word $w = abbbab$

Dynamic membership under substitution updates

Say we have $L = (ab)^*$ and we read a word $w = abbbab$

- Substitution update: set $w[i] := x$
  - $\rightarrow$ Can be done in $O(1)$ with an array
  - $\rightarrow$ The length n never changes

Dynamic membership under substitution updates

Say we have $L = (ab)^*$ and we read a word $w = abbbab$

- Substitution update: set $w[i] := x$
  - $\rightarrow$ Can be done in $O(1)$ with an array
  - $\rightarrow$ The length n never changes
- Maintain the membership of w to L under substitution updates
  - $\rightarrow$ Initially, we have $w \notin L$
  - $\rightarrow$ After $w[3] := a$: we have $w \in L$

Dynamic membership under substitution updates

Say we have $L = (ab)^*$ and we read a word $w = abbbab$

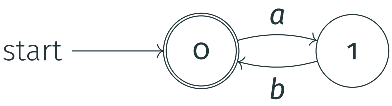
- Substitution update: set $w[i] := x$
  - $\rightarrow$ Can be done in $O(1)$ with an array
  - $\rightarrow$ The length n never changes
- Maintain the membership of w to L under substitution updates
  - $\rightarrow$ Initially, we have $w \notin L$
  - $\rightarrow$ After $w[3] := a$: we have $w \in L$

Theorem

For any regular language L recognized by an NFA A, given a word w, we can maintain dynamic membership of w to L under substitution updates in $O(Poly(|A|) \times \log |w|)$ combined complexity per update

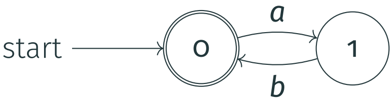
Proof sketch of the $O(\log n)$ algorithm

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



Proof sketch of the $O(\log n)$ algorithm

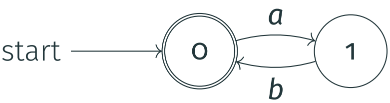
Fix the language $L = (ab)^*$:



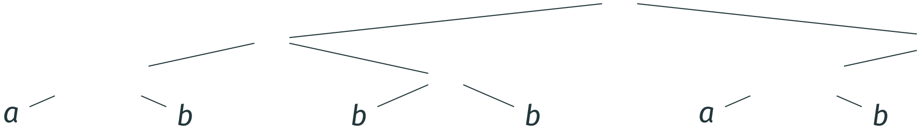
- Build a **balanced binary tree** on the input word $w = abbbab$

Proof sketch of the $O(\log n)$ algorithm

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

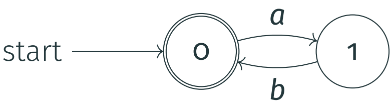


- Build a balanced binary tree on the input word $w = abbbab$

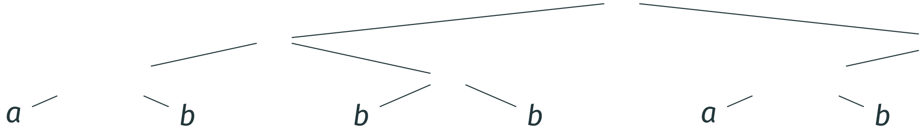


Proof sketch of the $O(\log n)$ algorithm

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

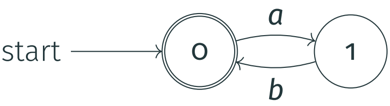


- Build a balanced binary tree on the input word $w = abbbab$
- Label each node n by the transition monoid element: all pairs $q \rightsquigarrow q'$ such that we can go from q to q' by reading the subword below n

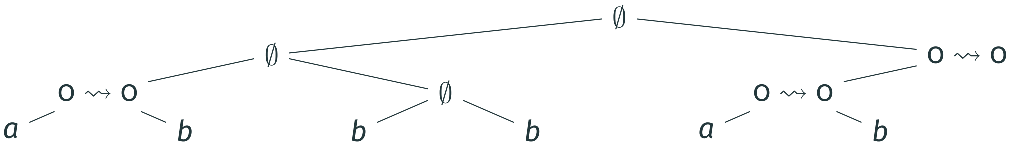


Proof sketch of the $O(\log n)$ algorithm

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

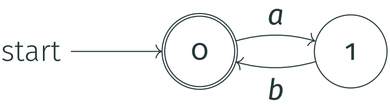


- Build a **balanced binary tree** on the input word $w = abbbab$
- Label each node n by the **transition monoid** element: all pairs $q \rightsquigarrow q'$ such that we can go from q to q' by reading the subword below n

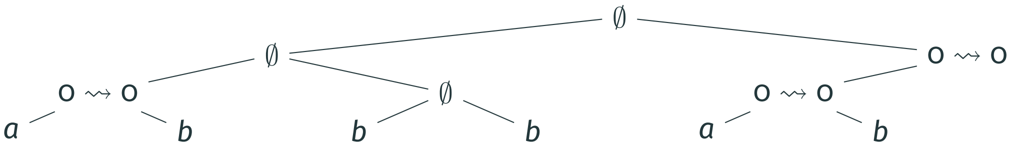


Proof sketch of the $O(\log n)$ algorithm

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



- Build a **balanced binary tree** on the input word $w = abbbab$
- 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 subword 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 **$O(\log n / \log \log n)$**

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 data complexity of dynamic membership, depending on the fixed regular language L?

Summary of our results [A., Jachiet, Paperman, 2021]

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 [A., Jachiet, Paperman, 2021]

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: [Skovbjerg Frandsen et al., 1997])

Summary of our results [A., Jachiet, Paperman, 2021]

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: [Skovbjerg Frandsen et al., 1997])
- The problem is always in $O(\log n / \log\log n)$

Summary of our results [A., Jachiet, Paperman, 2021]

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: [Skovbjerg Frandsen et al., 1997])
- The problem is always in $O(\log n / \log\log n)$

Generalizations to trees: [A., Barloy, Jachiet, Paperman, 2025] and Labbé's PhD

Enumeration of Factors

Enumeration of factors

- Membership of a word w to a language L is a Boolean query: yes/no answer
- What if we want to find the matches of L in w?

Enumeration of factors

- Membership of a word w to a language L is a Boolean query: yes/no answer
- What if we want to find the matches of L in w?

Factor enumeration problem:

- **Fix:** regular language L
- **Input:** word $w = a_1 \cdots a_n$
- **Output:** enumerate all pairs $[i, j\rangle$ such that $a_i \cdots a_{j-1} \in L$

$\rightarrow$ This is like `re.findall` but returning all pairs (including overlapping ones)

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

```
l o l
```

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

$[\rangle$ l o l

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

$[$ `l` $\rangle$ `o` `l`

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

$[$ l o $\rangle$ l

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

$[$ l o l $\rangle$

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

l $[\rangle$ o l

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

l $[$ o $\rangle$ l

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

`1` $[$ `o` `1` $\rangle$

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

l o [⟩ l

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

l o [l $\rangle$

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

`l o l` $[\rangle$

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

```
    l   o   l
```

$\rightarrow$ Complexity in w is $O(|w|^2 \times |w|)$

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

`l o l`

$\rightarrow$ Complexity in w is $O(|w|^2 \times |w|)$

$\rightarrow$ Can be optimized to $O(|w|^2)$

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

```
    l   o   l
```

  → Complexity in w is $O(|w|^2 \times |w|)$
  → Can be optimized to $O(|w|^2)$

- Problem: We may need to output $\Omega(|w|^2)$ pairs:

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

```
    l   o   l
```

  → Complexity in w is $O(|w|^2 \times |w|)$
  → Can be optimized to $O(|w|^2)$

- Problem: We may need to output $\Omega(|w|^2)$ pairs:
  - For the language $L = a^*$ on $w = a^n$, the number of matches is $\Omega(|w|^2)$

Measuring the complexity

- Naive algorithm: Run a DFA A for L on each factor of w

```
    l   o   l
```

→ Complexity in w is $O(|w|^2 \times |w|)$

→ Can be optimized to $O(|w|^2)$

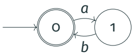
- Problem: We may need to output $\Omega(|w|^2)$ pairs:
  - For the language $L = a^*$ on $w = a^n$, the number of matches is $\Omega(|w|^2)$

→ We need to measure complexity differently

Enumeration algorithms

```
Antoine Amarilli Description Name Antoine
Amarilli.  Handle:  a3nm.  Identity Born
1990-02-07.  French national.  Appearance as
of 2017.  Auth OpenPGP. OpenId.  Bitcoin.
Contact Email and XMPP a3nm@a3nm.net
Affiliation Associate professor ...
```

Word w

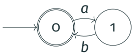


Automaton A

Enumeration algorithms

Antoine Amarilli Description Name Antoine Amarilli. Handle: a3nm. Identity Born 1990-02-07. French national. Appearance as of 2017. Auth OpenPGP. OpenId. Bitcoin. Contact Email and XMPP a3nm@a3nm.net Affiliation Associate professor ...

Word w

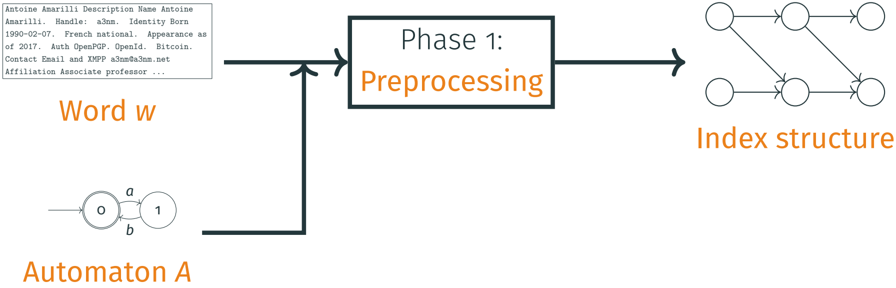


Automaton A

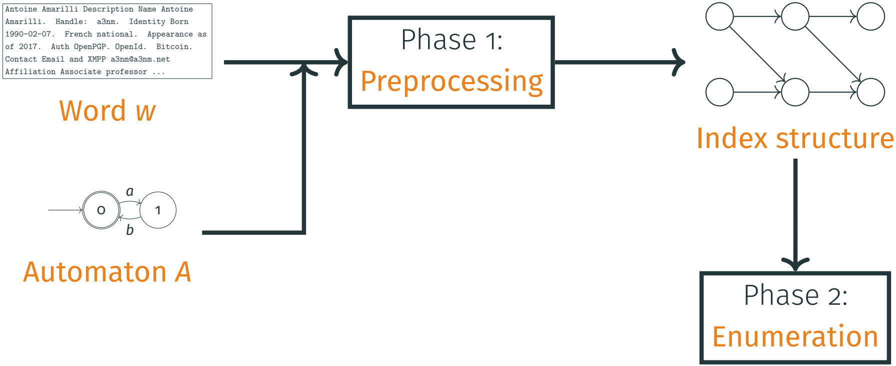
Phase 1:

Preprocessing

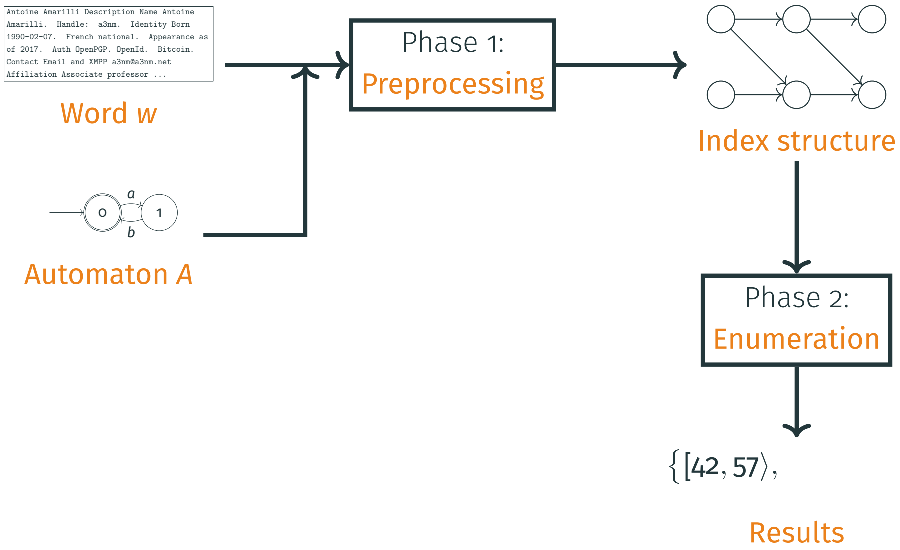
Enumeration algorithms



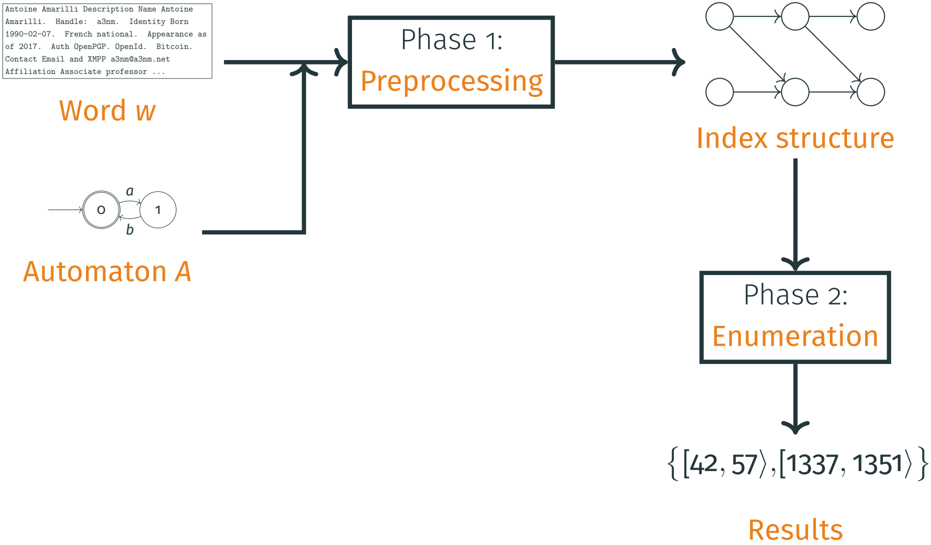
Enumeration algorithms



Enumeration algorithms



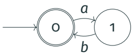
Enumeration algorithms



Enumeration algorithms

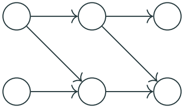
Antoine Amarilli Description Name Antoine Amarilli. Handle: a3nm. Identity Born 1990-02-07. French national. Appearance as of 2017. Auth OpenPGP. OpenId. Bitcoin. Contact Email and XMPP a3nm@a3nm.net Affiliation Associate professor ...

Word w



Automaton A

Phase 1: Preprocessing



Index structure

Phase 2: Enumeration

$\{[42, 57\rangle, [1337, 1351\rangle\}$

Results

Two ways to measure performance:

- Total time for phase 1
- Delay between two results in phase 2

... as a function of the word and automaton

Results for factor enumeration

Existing work has shown the best possible bounds :

Theorem (follows from [Florenzano et al., 2018])

Given a word w and a DFA A, we can enumerate the factors in w that match A with preprocessing $O(|w| \times |A|)$ and delay $O(1)$.

Results for factor enumeration

Existing work has shown the best possible bounds for DFAs:

Theorem (follows from [Florenzano et al., 2018])
Given a word w and a DFA A, we can enumerate the factors in w that match A with preprocessing $O(|w| \times |A|)$ and delay $O(1)$.

What if the automaton is nondeterministic?

Results for factor enumeration

Existing work has shown the best possible bounds for DFAs:

Theorem (follows from [Florenzano et al., 2018])

Given a word w and a DFA A, we can enumerate the factors in w that match A with preprocessing $O(|w| \times |A|)$ and delay $O(1)$.

What if the automaton is nondeterministic?

Theorem ([A., Bourhis, Mengel, Niewerth, 2019a])

For a NFA A, we can enumerate the factors of w that match A with preprocessing $O(|w| \times Poly(|A|))$ and delay $O(Poly(|A|))$.

Automata with captures

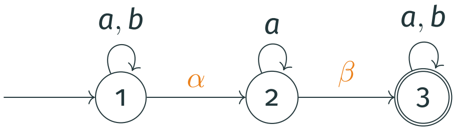
The result extends beyond factor enumeration to automata with captures

$\rightarrow$ Example: $(a|b)^* \ \alpha \ a^* \ \beta \ (a|b)^*$

Automata with captures

The result extends beyond factor enumeration to automata with captures

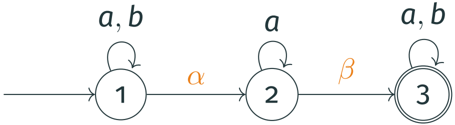
→ Example: $(a|b)^* \; \alpha \; a^* \; \beta \; (a|b)^*$



Automata with captures

The result extends beyond factor enumeration to automata with captures

$\rightarrow$ Example: $(a|b)^* \ \alpha \ a^* \ \beta \ (a|b)^*$



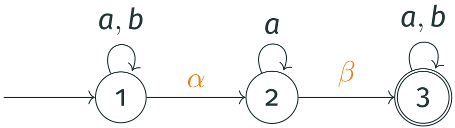
Semantics of the automaton A:

- Reads letters from the text
- Guesses matches of α and β at positions in the text

Automata with captures

The result extends beyond factor enumeration to automata with captures

→ Example: $(a|b)^* \; \alpha \; a^* \; \beta \; (a|b)^*$



Semantics of the automaton A:

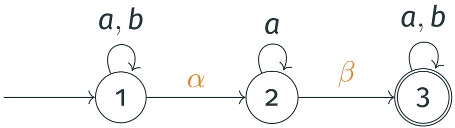
- Reads letters from the text
- Guesses matches of α and β at positions in the text

→ Output: tuples $\langle \alpha : i, \beta : j \rangle$ such that
A has an accepting run reading α at position i and β at j

Automata with captures

The result extends beyond factor enumeration to automata with captures

→ Example: $(a|b)^* \; \alpha \; a^* \; \beta \; (a|b)^*$



Semantics of the automaton A:

- Reads letters from the text
- Guesses matches of α and β at positions in the text

→ Output: tuples $\langle \alpha : i, \beta : j \rangle$ such that
A has an accepting run reading α at position i and β at j

Also: generalizations to tree automata [A., Bourhis, Mengel, Niewerth, 2019b] and context-free grammars: [A., Jachiet, Muñoz, Riveros, 2022]

Proof idea: Product DAG

Compute a product DAG of the word W and of the automaton A

Proof idea: Product DAG

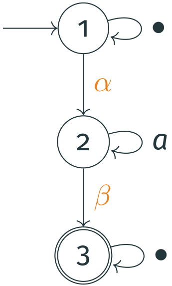
Compute a product DAG of the word W and of the automaton A

Example: Word $w := aaaba$ and $P := \bullet^* \;\; \alpha \;\; a^* \;\; \beta \;\; \bullet^*$,

Proof idea: Product DAG

Compute a product DAG of the word W and of the automaton A

Example: Word $w := aaaba$ and $P := \bullet^* \ \alpha \ a^* \ \beta \ \bullet^*$,

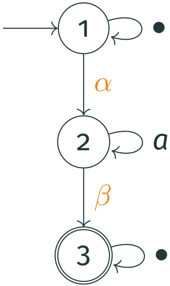


Proof idea: Product DAG

Compute a product DAG of the word W and of the automaton A

Example: Word $w := aaaba$ and $P := \bullet^* \ \alpha \ a^* \ \beta \ \bullet^*$,

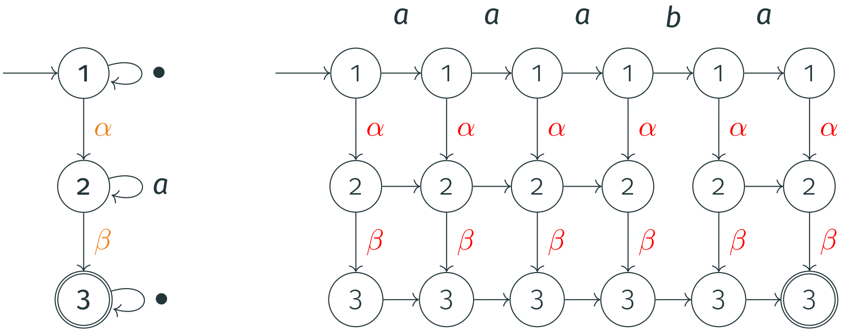
$a \quad a \quad a \quad b \quad a$



Proof idea: Product DAG

Compute a product DAG of the word W and of the automaton A

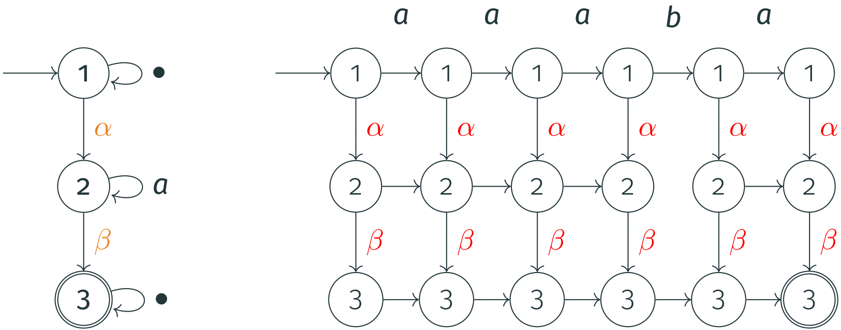
Example: Word $w := aaaba$ and $P := \bullet^* \ \alpha \ a^* \ \beta \ \bullet^*$,



Proof idea: Product DAG

Compute a product DAG of the word W and of the automaton A

Example: Word $w := aaaba$ and $P := \bullet^* \ \alpha \ a^* \ \beta \ \bullet^*$,

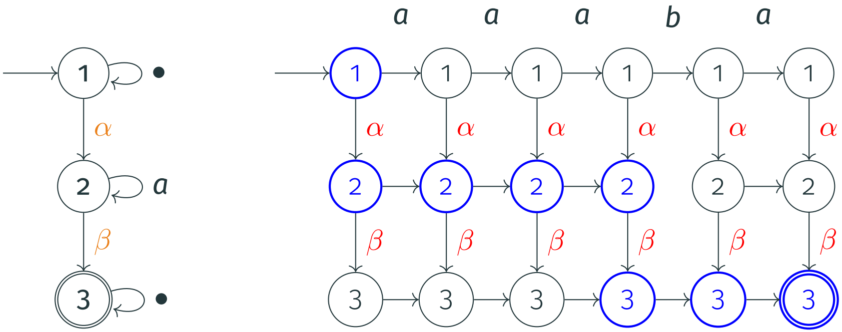


$\rightarrow$ Each path in the product DAG corresponds to a match

Proof idea: Product DAG

Compute a product DAG of the word W and of the automaton A

Example: Word $w := aaaba$ and $P := \bullet^* \ \alpha \ a^* \ \beta \ \bullet^*$, match $\langle \alpha : 0, \beta : 3 \rangle$

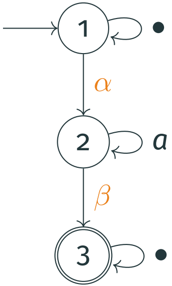


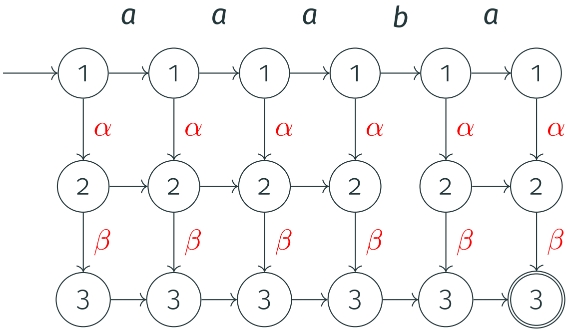
→ Each path in the product DAG corresponds to a match

Proof idea: Product DAG

Compute a product DAG of the word W and of the automaton A

Example: Word $w := aaaba$ and $P := \bullet^* \ \alpha \ a^* \ \beta \ \bullet^*$,





$\rightarrow$ Each path in the product DAG corresponds to a match

Challenge: Enumerate paths but avoid duplicate matches

Regular Languages on Graphs

Regular Path Queries (RPQs)

- Fix a finite alphabet Σ

Regular Path Queries (RPQs)

- Fix a finite alphabet Σ

$$\Sigma = \{a, b\}$$

Regular Path Queries (RPQs)

- Fix a finite **alphabet** Σ
- **Regular path query** RPQ_L
  - $\rightarrow$ Given by a **regular language** L on Σ

$$\Sigma = \{a, b\}$$

Regular Path Queries (RPQs)

- Fix a finite alphabet Σ
- Regular path query RPQ_L
  - $\rightarrow$ Given by a regular language L on Σ

$$\Sigma = \{a, b\}$$

$$L = a^* \, b \, a^*$$

Regular Path Queries (RPQs)

- Fix a finite alphabet Σ
- Regular path query RPQ_L
  - $\rightarrow$ Given by a regular language L on Σ
- Graph database $D = (V, E)$
  - $\rightarrow$ Vertices V and edges $E \subseteq V \times \Sigma \times V$

$$\Sigma = \{a, b\}$$

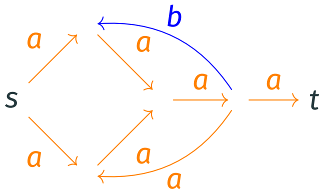
$$L = a^* \, b \, a^*$$

Regular Path Queries (RPQs)

- Fix a finite alphabet Σ
- Regular path query RPQ_L
  → Given by a regular language L on Σ
- Graph database $D = (V, E)$
  → Vertices V and edges $E \subseteq V \times \Sigma \times V$

$\Sigma = \{a, b\}$

$L = a^* \, b \, a^*$

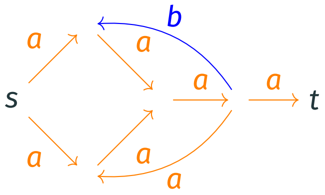


Regular Path Queries (RPQs)

- Fix a finite alphabet Σ
- Regular path query RPQ_L
  - $\rightarrow$ Given by a regular language L on Σ
- Graph database $D = (V, E)$
  - $\rightarrow$ Vertices V and edges $E \subseteq V \times \Sigma \times V$
- We have $D \models \mathrm{RPQ}_L(s, t)$ for endpoints s, t if:
  - We have a walk w in D from the source s to the target t
  - The concatenation of the edge labels of w is in L

$\Sigma = \{a, b\}$

$L = a^* \, b \, a^*$

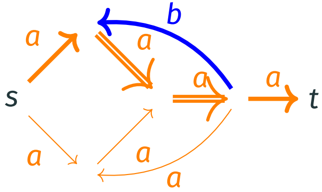


Regular Path Queries (RPQs)

- Fix a finite alphabet Σ
- Regular path query RPQ_L
  - $\rightarrow$ Given by a regular language L on Σ
- Graph database $D = (V, E)$
  - $\rightarrow$ Vertices V and edges $E \subseteq V \times \Sigma \times V$
- We have $D \models \mathrm{RPQ}_L(s, t)$ for endpoints s, t if:
  - We have a walk w in D from the source s to the target t
  - The concatenation of the edge labels of w is in L

$\Sigma = \{a, b\}$

$L = a^* \, b \, a^*$

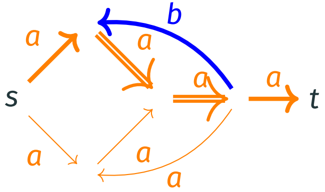


Regular Path Queries (RPQs)

- Fix a finite alphabet Σ
- Regular path query RPQ_L
  - $\rightarrow$ Given by a regular language L on Σ
- Graph database $D = (V, E)$
  - $\rightarrow$ Vertices V and edges $E \subseteq V \times \Sigma \times V$
- We have $D \models \mathrm{RPQ}_L(s, t)$ for endpoints s, t if:
  - We have a walk w in D from the source s to the target t
  - The concatenation of the edge labels of w is in L
  - Note: w is not necessarily a simple path

$\Sigma = \{a, b\}$

$L = a^* \, b \, a^*$



Evaluating RPQs on graph databases

- **Fix:** a **regular language** L giving the regular path query RPQ_L
- **Input** a **graph database** D, source s, target t
- **Output:** decide whether $D \models \mathrm{RPQ}_L(s, t)$

Evaluating RPQs on graph databases

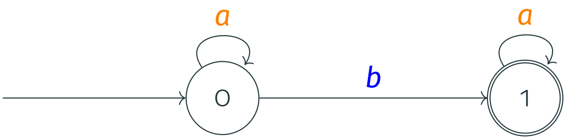
- **Fix:** a **regular language** L giving the regular path query RPQ_L
- **Input** a **graph database** D, source s, target t
- **Output:** decide whether $D \models \mathrm{RPQ}_L(s, t)$

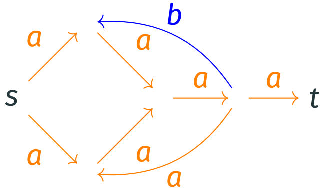
This can be solved...

Evaluating RPQs on graph databases

- **Fix:** a **regular language** L giving the regular path query RPQ_L
- **Input** a **graph database** D, source s, target t
- **Output:** decide whether $D \models \mathrm{RPQ}_L(s, t)$

This can be solved... **in PTIME** via a **product construction**: (or via Datalog)

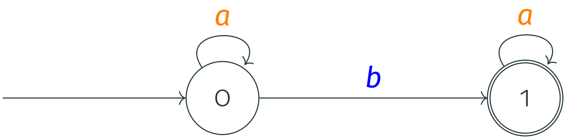


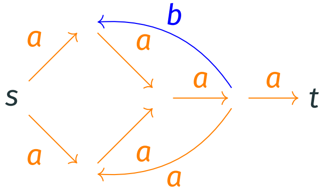


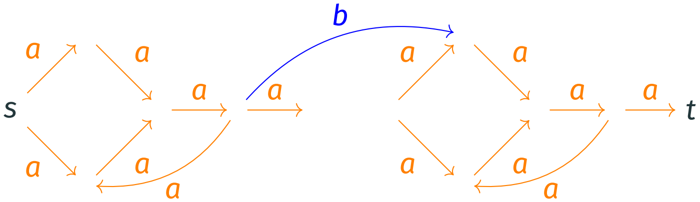
Evaluating RPQs on graph databases

- **Fix:** a regular language L giving the regular path query RPQ_L
- **Input** a graph database D, source s, target t
- **Output:** decide whether $D \models \mathrm{RPQ}_L(s,t)$

This can be solved... in PTIME via a product construction: (or via Datalog)







Generalizations of RPQs

We have defined the RPQ evaluation problem, which generalizes regular language membership to graph databases:

- Fix: a regular language L giving the regular path query RPQ_L
- Input a graph database D, source s, target t
- Output: decide whether $D \models \mathrm{RPQ}_L(s, t)$

Generalizations of RPQs

We have defined the RPQ evaluation problem, which generalizes regular language membership to graph databases:

- Fix: a regular language L giving the regular path query RPQ_L
- Input a graph database D, source s, target t
- Output: decide whether $D \models \mathrm{RPQ}_L(s, t)$

Many possible generalizations of RPQ evaluation:

- I will focus on one: the smallest witness problem
  - $\rightarrow$ *What is the smallest sub-database of D that satisfies RPQ_L?*

Generalizations of RPQs

We have defined the RPQ evaluation problem, which generalizes regular language membership to graph databases:

- Fix: a regular language L giving the regular path query RPQ_L
- Input a graph database D, source s, target t
- Output: decide whether $D \models \mathrm{RPQ}_L(s,t)$

Many possible generalizations of RPQ evaluation:

- I will focus on one: the smallest witness problem
  - $\rightarrow$ *What is the smallest sub-database of D that satisfies RPQ_L?*
- I will briefly mention a related problem: resilience

Smallest Witness for RPQs

- Decision problem SW_L for a fixed regular language L:
    - Input: graph database D, vertices s and t, integer $k \in \mathbb{N}$

Smallest Witness for RPQs

- Decision problem SW_L for a fixed regular language L:
  - Input: graph database D, vertices s and t, integer $k \in \mathbb{N}$
  - Output: is there a subdatabase $D' \subseteq D$ with $\leq k$ edges with $D' \models \mathsf{RPQ}_L(s, t)$

Smallest Witness for RPQs

- Decision problem SW_L for a fixed regular language L:
  - Input: graph database D, vertices s and t, integer $k \in \mathbb{N}$
  - Output: is there a subdatabase $D' \subseteq D$ with $\leq k$ edges with $D' \models \mathrm{RPQ}_L(s, t)$

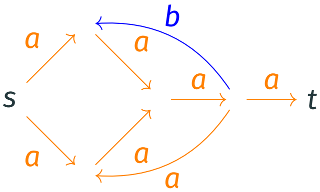
$\rightarrow$ What is the complexity of SW_L depending on the language L?

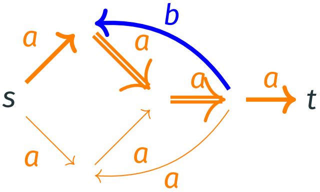
Smallest Witness for RPQs

- Decision problem SW_L for a fixed regular language L:
  - Input: graph database D, vertices s and t, integer $k \in \mathbb{N}$
  - Output: is there a subdatabase $D' \subseteq D$ with $\leq k$ edges with $D' \models \mathsf{RPQ}_L(s, t)$

$\rightarrow$ What is the complexity of SW_L depending on the language L?

For instance for $L = a^* \, b \, a^*$ on the example database...



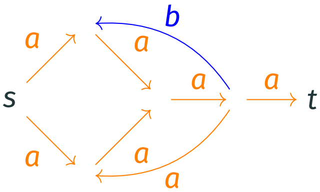


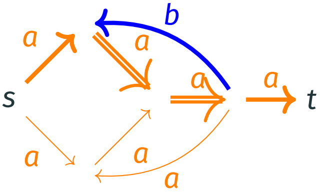
Smallest Witness for RPQs

- Decision problem SW_L for a fixed regular language L:
  - Input: graph database D, vertices s and t, integer $k \in \mathbb{N}$
  - Output: is there a subdatabase $D' \subseteq D$ with $\leq k$ edges with $D' \models \mathsf{RPQ}_L(s, t)$

$\rightarrow$ What is the complexity of SW_L depending on the language L?

For instance for $L = a^* \, b \, a^*$ on the example database... true iff $k \geq 5$





Smallest Witness complexity

First results:

- SW_L is always **in NP**
  - $\rightarrow$ **Guess** the subdatabase D' with $|D'| \leq k$ and **verify** $D' \models \mathsf{RPQ}_L(s, t)$ in PTIME

Smallest Witness complexity

First results:

- SW_L is always **in NP**
  - $\rightarrow$ **Guess** the subdatabase D' with $|D'| \leq k$ and **verify** $D' \models \mathsf{RPQ}_L(s, t)$ in PTIME
- If L is a **finite language**, then SW_L is **in PTIME**
  - $\rightarrow$ Polynomial number of possible matches so we can **bruteforce**

Smallest Witness complexity

First results:

- SW_L is always **in NP**
  - $\rightarrow$ **Guess** the subdatabase D' with $|D'| \leq k$ and **verify** $D' \models \mathsf{RPQ}_L(s, t)$ in PTIME
- If L is a **finite language**, then SW_L is **in PTIME**
  - $\rightarrow$ Polynomial number of possible matches so we can **bruteforce**
- For $L = a^*$, we have that SW_L is...

Smallest Witness complexity

First results:

- SW_L is always **in NP**
  - $\rightarrow$ Guess the subdatabase D' with $|D'| \leq k$ and verify $D' \models \mathsf{RPQ}_L(s, t)$ in PTIME
- If L is a finite language, then SW_L is in PTIME
  - $\rightarrow$ Polynomial number of possible matches so we can bruteforce
- For $L = a^*$, we have that SW_L is... in PTIME
  - $\rightarrow$ Compute the shortest path from s to t and check that it has $\leq k$ edges

Smallest Witness for parity constraints

What about $L = (aa)^*$?

Smallest Witness for parity constraints

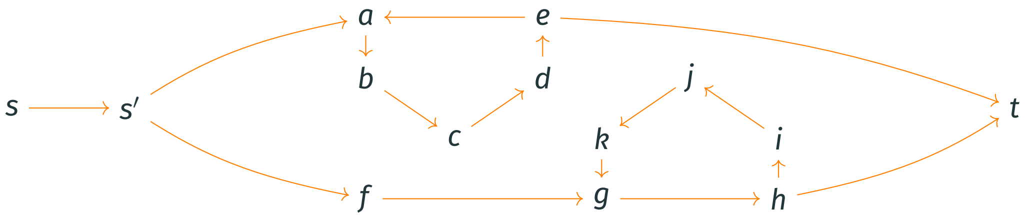
What about $L = (aa)^*$?

"Given a graph G, source s, and target t, find an st-walk of even length that uses the least number of distinct edges"

Smallest Witness for parity constraints

What about $L = (aa)^*$?

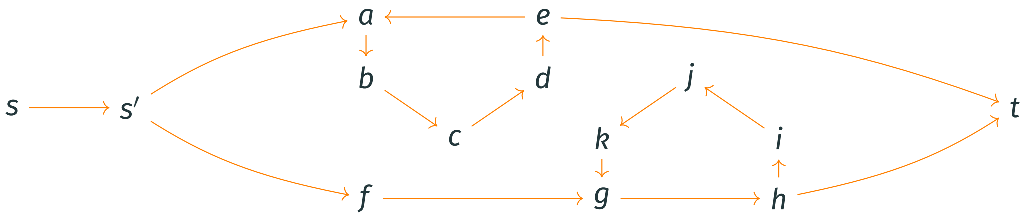
"Given a graph G, source s, and target t, find an st-walk of even length that uses the least number of distinct edges"



Smallest Witness for parity constraints

What about $L = (aa)^*$?

"Given a graph G, source s, and target t, find an st-walk of even length that uses the least number of distinct edges"

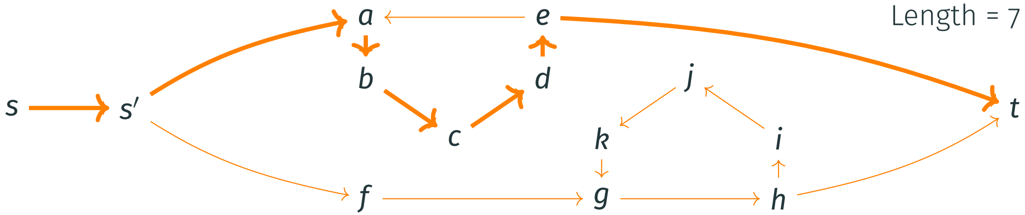


- It may not be a simple path,

Smallest Witness for parity constraints

What about $L = (aa)^*$?

"Given a graph G, source s, and target t, find an st-walk of even length that uses the least number of distinct edges"

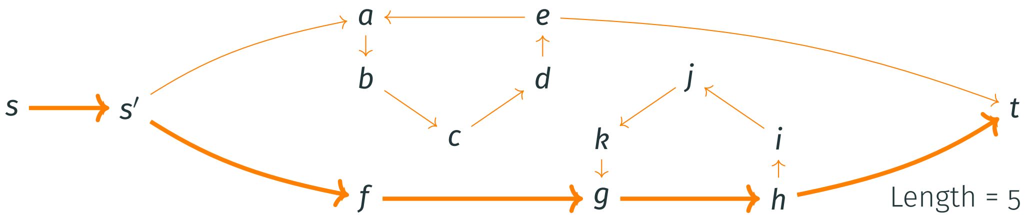


- It may not be a simple path,

Smallest Witness for parity constraints

What about $L = (aa)^*$?

"Given a graph G, source s, and target t, find an st-walk of even length that uses the least number of distinct edges"

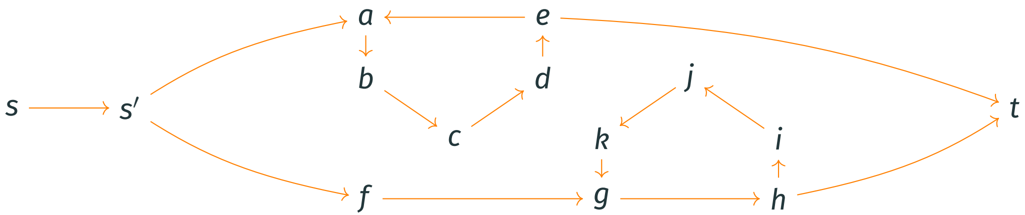


- It may not be a simple path,

Smallest Witness for parity constraints

What about $L = (aa)^*$?

"Given a graph G, source s, and target t, find an st-walk of **even length** that uses the least number of **distinct edges**"

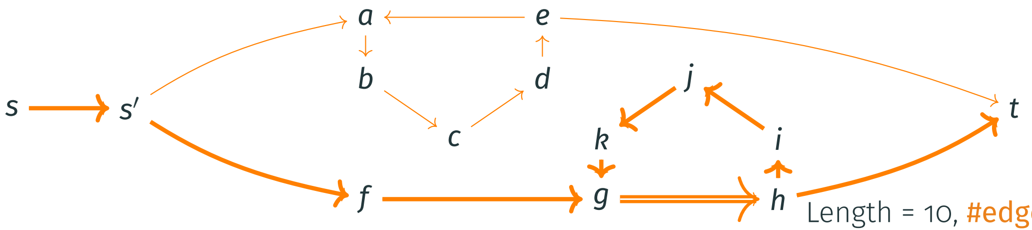


- It may not be a **simple path**, and it may not be a **shortest walk**!

Smallest Witness for parity constraints

What about $L = (aa)^*$?

"Given a graph G, source s, and target t, find an st-walk of **even length** that uses the least number of **distinct edges**"



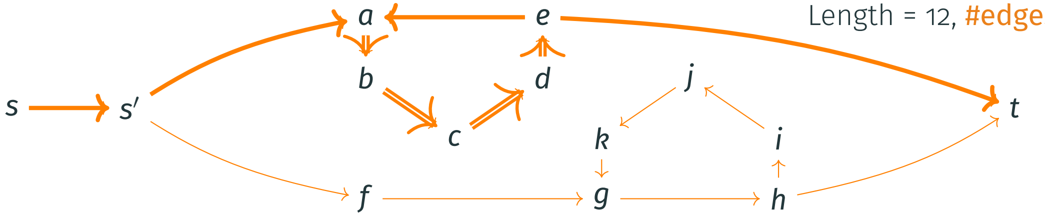
Length = 10, #edges = 9

- It may not be a **simple path**, and it may not be a **shortest walk**!

Smallest Witness for parity constraints

What about $L = (aa)^*$?

"Given a graph G, source s, and target t, find an st-walk of even length that uses the least number of distinct edges"

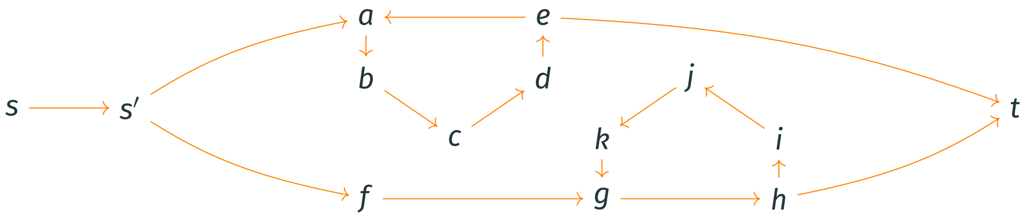


- It may not be a simple path, and it may not be a shortest walk!

Smallest Witness for parity constraints

What about $L = (\textcolor{orange}{aa})^*$?

"Given a graph G, source s, and target t, find an st-walk of even length that uses the least number of distinct edges"



- It may not be a simple path, and it may not be a shortest walk!
- Is this problem in PTIME?

Tractability for modularity constraints

$$L = (a^q)^* a^r$$

Tractability for modularity constraints

$L = (a^q)^* a^r$ (*st*-walk of length $r \bmod q$ with min. #distinct edges)

Tractability for modularity constraints

$L = (a^q)^* a^r$ (*st*-walk of length $r \bmod q$ with min. #distinct edges)

Theorem ([A., Groz, Wein, 2025])

For any fixed $q > 0$ and $0 \leq r < q$, letting $L = (a^q)^* a^r$, the problem SW_L is in PTIME

Tractability for modularity constraints

$$L = (a^q)^* a^r \qquad (st\text{-walk of length } r \bmod q \text{ with min. \#distinct edges})$$

Theorem ([A., Groz, Wein, 2025])

For any fixed $q > 0$ and $0 \leq r < q$, letting $L = (a^q)^* a^r$, the problem SW_L is in PTIME

Proof sketch:

- An optimal walk w will not have too many **detours**

Tractability for modularity constraints

$$L = (a^q)^* a^r \qquad (st\text{-walk of length } r \bmod q \text{ with min. \#distinct edges})$$

Theorem ([A., Groz, Wein, 2025])

For any fixed $q > 0$ and $0 \leq r < q$, letting $L = (a^q)^* a^r$, the problem SW_L is in PTIME

Proof sketch:

- An optimal walk w will not have too many detours
  - Detour: taking a new edge (u, v) then going back to a vertex already reachable from u

Tractability for modularity constraints

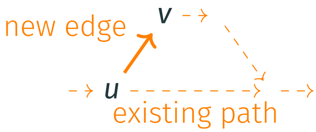
$L = (a^q)^* a^r$ (st-walk of length $r \bmod q$ with min. #distinct edges)

Theorem ([A., Groz, Wein, 2025])

For any fixed $q > 0$ *and* $0 \leq r < q$, *letting* $L = (a^q)^* a^r$, *the problem* SW_L *is* ***in PTIME***

Proof sketch:

- An optimal walk w will not have too many **detours**
  - **Detour**: taking a new edge (u, v) then going back to a vertex already reachable from u



Tractability for modularity constraints

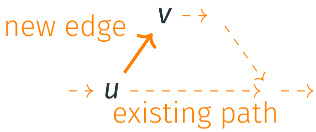
$L = (a^q)^* a^r$ (st-walk of length r mod q with min. #distinct edges)

Theorem ([A., Groz, Wein, 2025])

For any fixed $q > 0$ and $0 \leq r < q$, letting $L = (a^q)^* a^r$, the problem SW_L is in PTIME

Proof sketch:

- An optimal walk w will not have too many **detours**
  - **Detour**: taking a new edge (u, v) then going back to a vertex already reachable from u
  - Only useful to **change the remainder** of the length



Tractability for modularity constraints

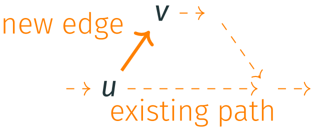
$L = (a^q)^* a^r$ (*st*-walk of length $r \bmod q$ with min. #distinct edges)

Theorem ([A., Groz, Wein, 2025])

For any fixed $q > 0$ and $0 \leq r < q$, letting $L = (a^q)^* a^r$, the problem SW_L is in PTIME

Proof sketch:

- An optimal walk w will not have too many **detours**
  - **Detour**: taking a new edge (u, v) then going back to a vertex already reachable from u
  - Only useful to **change the remainder** of the length
  - After $O(\log q)$ detours, **all possible remainders** achieved



Tractability for modularity constraints

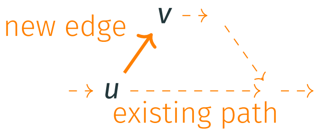
$$L = (a^q)^* a^r \qquad (st\text{-walk of length } r \bmod q \text{ with min. \#distinct edges})$$

Theorem ([A., Groz, Wein, 2025])

For any fixed $q > 0$ and $0 \leq r < q$, letting $L = (a^q)^* a^r$, the problem SW_L is in PTIME

Proof sketch:

- An optimal walk w will not have too many detours
  - Detour: taking a new edge (u, v) then going back to a vertex already reachable from u
  - Only useful to change the remainder of the length
  - After $O(\log q)$ detours, all possible remainders achieved



- The cutwidth of a graph spanned by walk w is bounded by $O(\#\text{detours of } w)$

Tractability for modularity constraints

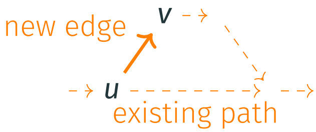
$L = (a^q)^* a^r$ (st-walk of length $r \bmod q$ with min. #distinct edges)

Theorem ([A., Groz, Wein, 2025])

For any fixed $q > 0$ and $0 \leq r < q$, letting $L = (a^q)^* a^r$, the problem SW_L is in PTIME

Proof sketch:

- An optimal walk w will not have too many **detours**
  - **Detour**: taking a new edge (u, v) then going back to a vertex already reachable from u
  - Only useful to **change the remainder** of the length
  - After $O(\log q)$ detours, **all possible remainders** achieved



- The **cutwidth** of a graph spanned by walk w is bounded by $O(\#\text{detours of } w)$
- Bounded-cutwidth subgraphs can be found by **dynamic programming**

An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$?

An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$? Then SW_L is **NP-complete** (from 3-SAT)

An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$? Then SW_L is **NP-complete** (from 3-SAT)

Variable guess gadget

s

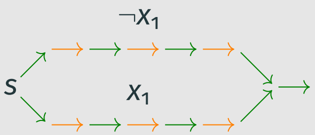
t

Clause check gadget

An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$?

Then SW_L is **NP-complete** (from 3-SAT)



Variable guess gadget

t

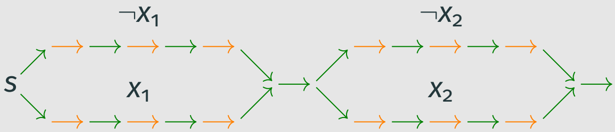
Clause check gadget

An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$?

Then SW_L is NP-complete (from 3-SAT)

Variable guess gadget

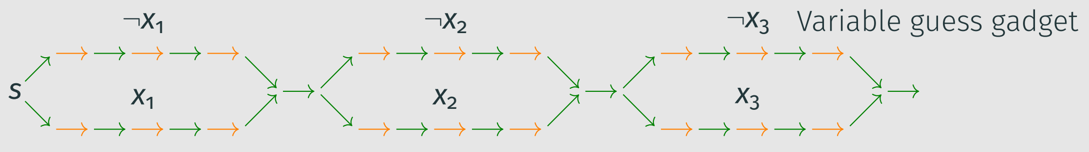


t

Clause check gadget

An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$? Then SW_L is **NP-complete** (from 3-SAT)



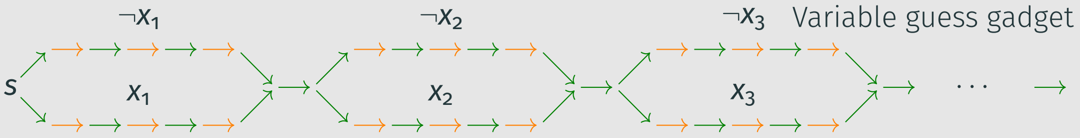
t

Clause check gadget

An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$?

Then SW_L is **NP-complete** (from 3-SAT)

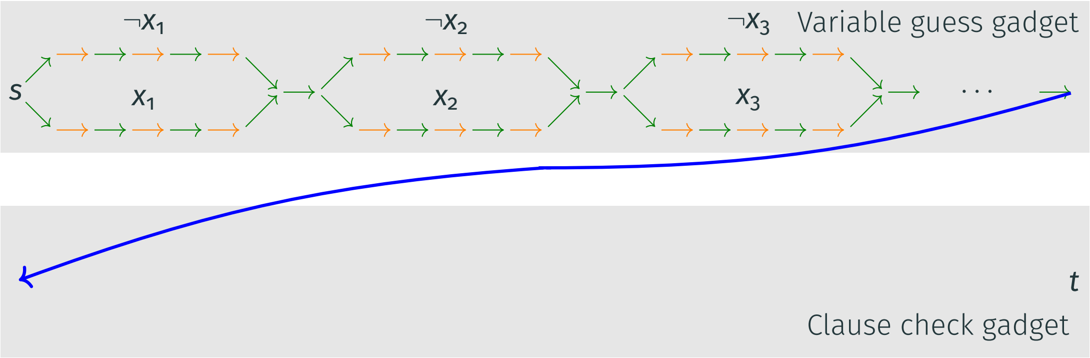


t

Clause check gadget

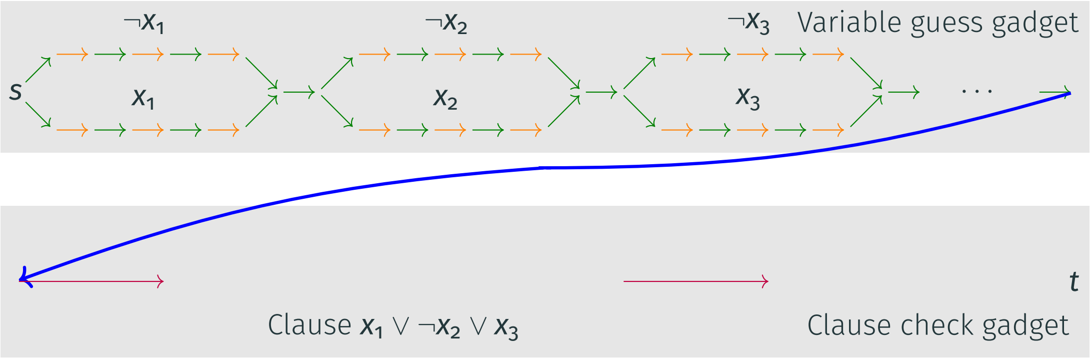
An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$?   Then SW_L is **NP-complete** (from 3-SAT)



An intractable case

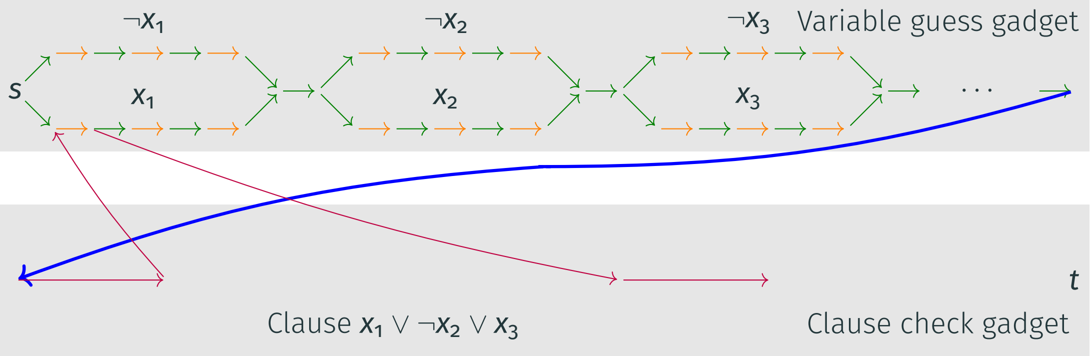
What about $L = (a + c)^* \, b \, (a + d)^*$? Then SW_L is **NP-complete** (from 3-SAT)



An intractable case

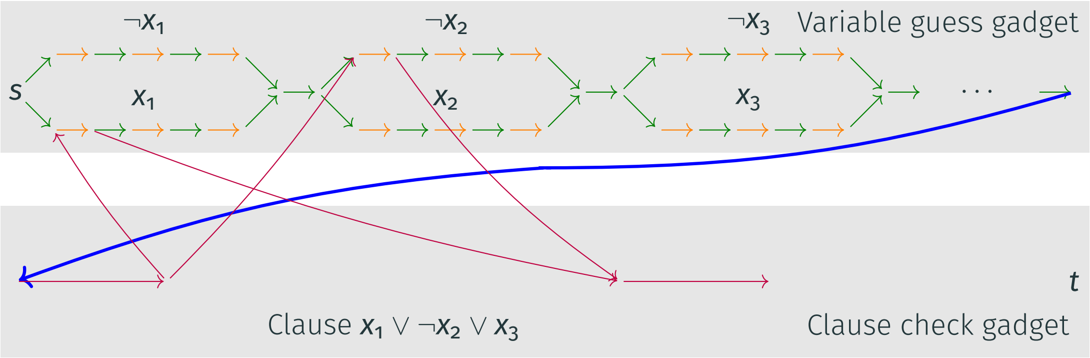
What about $L = (a + c)^* \, b \, (a + d)^*$?

Then SW_L is **NP-complete** (from 3-SAT)



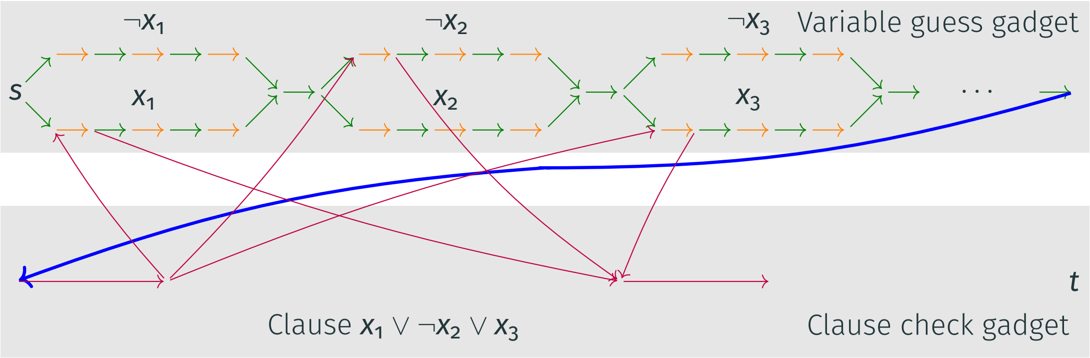
An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$? Then SW_L is **NP-complete** (from 3-SAT)



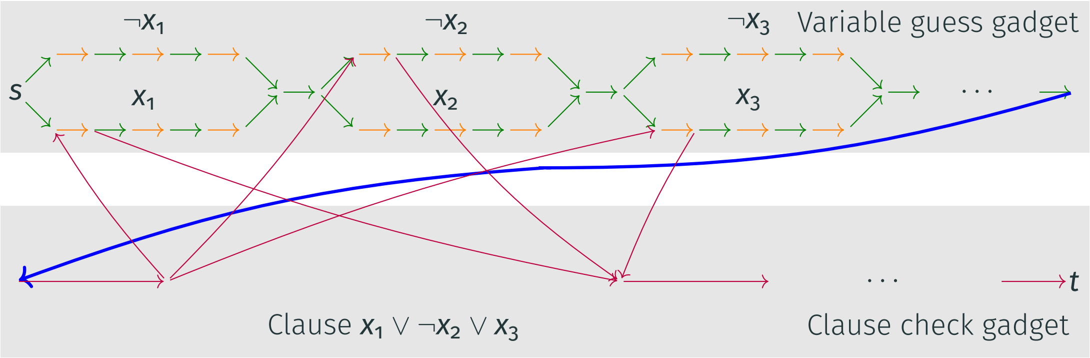
An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$? Then SW_L is **NP-complete** (from 3-SAT)



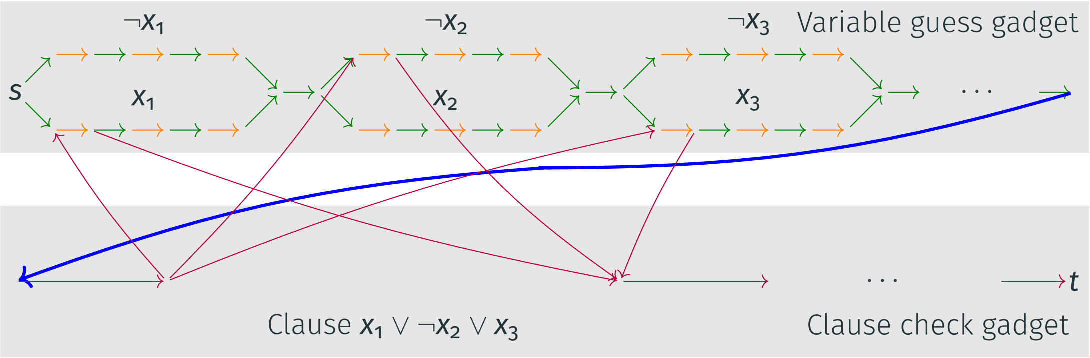
An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$? Then SW_L is **NP-complete** (from 3-SAT)



An intractable case

What about $L = (a + c)^* \, b \, (a + d)^*$? Then SW_L is **NP-complete** (from 3-SAT)



Minimizing #distinct edges forces that all a-edges in the clause checks are revisits

Resilience

Fix an RPQ Q, read as input a database D

- Smallest Witness for Q: a subdatabase $D' \subseteq D$ of minimum size with $D' \models Q$

Resilience

Fix an RPQ Q, read as input a database D

- Smallest Witness for Q: a subdatabase $D' \subseteq D$ of minimum size with $D' \models Q$
- Resilience for Q: a subdatabase $D' \subseteq D$ of maximum size with $D' \not\models Q$

Resilience

Fix an RPQ Q, read as input a database D

- Smallest Witness for Q: a subdatabase $D' \subseteq D$ of minimum size with $D' \models Q$
- Resilience for Q: a subdatabase $D' \subseteq D$ of maximum size with $D' \not\models Q$

What is the complexity of computing resilience for RPQs?

Resilience

Fix an RPQ Q, read as input a database D

- Smallest Witness for Q: a subdatabase $D' \subseteq D$ of minimum size with $D' \models Q$
- Resilience for Q: a subdatabase $D' \subseteq D$ of maximum size with $D' \not\models Q$

What is the complexity of computing resilience for RPQs?

- Can be in PTIME if the language of the RPQ is local (e.g., ab^*c)
  - $\rightarrow$ Reduces to a mincut problem
- Can be NP-hard, e.g., for $L = aa$ (in the setting without specified source/sink)
  - $\rightarrow$ Extends to any finite language featuring a word with a repeated letter
  - $\rightarrow$ Also other cases (languages which are non-local in a strong sense)
- Dichotomy for (2)RPQs via VCSPs [Bodirsky et al., 2024]
  - Applies to databases with weighted facts, and opaque (but decidable) criterion

Resilience

Fix an RPQ Q, read as input a database D

- Smallest Witness for Q: a subdatabase $D' \subseteq D$ of minimum size with $D' \models Q$
- Resilience for Q: a subdatabase $D' \subseteq D$ of maximum size with $D' \not\models Q$

What is the complexity of computing resilience for RPQs?

- Can be in PTIME if the language of the RPQ is local (e.g., ab^*c)
  - $\rightarrow$ Reduces to a mincut problem
- Can be NP-hard, e.g., for $L = aa$ (in the setting without specified source/sink)
  - $\rightarrow$ Extends to any finite language featuring a word with a repeated letter
  - $\rightarrow$ Also other cases (languages which are non-local in a strong sense)
- Dichotomy for (2)RPQs via VCSPs [Bodirsky et al., 2024]
  - Applies to databases with weighted facts, and opaque (but decidable) criterion

$\rightarrow$ Unknown if there is a unified understanding of these two problems

Resilience

Fix an RPQ Q, read as input a database D

- Smallest Witness for Q: a subdatabase $D' \subseteq D$ of minimum size with $D' \models Q$
- Resilience for Q: a subdatabase $D' \subseteq D$ of maximum size with $D' \not\models Q$

What is the complexity of computing resilience for RPQs?

- Can be in PTIME if the language of the RPQ is local (e.g., ab^*c)
  - $\rightarrow$ Reduces to a mincut problem
- Can be NP-hard, e.g., for $L = aa$ (in the setting without specified source/sink)
  - $\rightarrow$ Extends to any finite language featuring a word with a repeated letter
  - $\rightarrow$ Also other cases (languages which are non-local in a strong sense)
- Dichotomy for (2)RPQs via VCSPs [Bodirsky et al., 2024]
  - Applies to databases with weighted facts, and opaque (but decidable) criterion

$\rightarrow$ Unknown if there is a unified understanding of these two problems

See [A., Gatterbauer, Makhija, Monet, 2025]

Other Directions and Future Work

Many other directions (talk to me to know more!)

Many generalizations of the membership problem of words to regular languages:

- Partial and probabilistic words
  - Many open cases in data complexity for CFGs, and in combined complexity

Many other directions (talk to me to know more!)

Many generalizations of the **membership problem** of words to regular languages:

- Partial and probabilistic words
  - Many open cases in **data complexity** for CFGs, and in **combined complexity**
- Dynamic membership
  - What about **other updates** (insert/delete, cut and paste...)
  - What about **trees**? **graphs**?

Many other directions (talk to me to know more!)

Many generalizations of the **membership problem** of words to regular languages:

- Partial and probabilistic words
  - Many open cases in **data complexity** for CFGs, and in **combined complexity**
- Dynamic membership
  - What about **other updates** (insert/delete, cut and paste...)
  - What about **trees**? **graphs**?
- Enumeration
  - What about **incremental maintenance** of enumeration structures?

    → [A., Bourhis, Mengel, Niewerth, 2019b], [A., Dziadek, Segoufin, 2025]

Many other directions (talk to me to know more!)

Many generalizations of the **membership problem** of words to regular languages:

- Partial and probabilistic words
  - Many open cases in **data complexity** for CFGs, and in **combined complexity**
- Dynamic membership
  - What about **other updates** (insert/delete, cut and paste...)
  - What about **trees**? **graphs**?
- Enumeration
  - What about **incremental maintenance** of enumeration structures?
    $\rightarrow$ [A., Bourhis, Mengel, Niewerth, 2019b], [A., Dziadek, Segoufin, 2025]
- RPQs on graphs
  - What about **simple paths**/**trails**? cf [Bagan et al., 2020] [Martens et al., 2023]
  - What about **RPQs on probabilistic graphs**? [A., van Bremen, Gaspard, Meel, 2025]

Many other directions (talk to me to know more!)

Many generalizations of the membership problem of words to regular languages:

- Partial and probabilistic words
  - Many open cases in data complexity for CFGs, and in combined complexity
- Dynamic membership
  - What about other updates (insert/delete, cut and paste...)
  - What about trees? graphs?
- Enumeration
  - What about incremental maintenance of enumeration structures?
    - → [A., Bourhis, Mengel, Niewerth, 2019b], [A., Dziadek, Segoufin, 2025]
- RPQs on graphs
  - What about simple paths/trails? cf [Bagan et al., 2020] [Martens et al., 2023]
  - What about RPQs on probabilistic graphs? [A., van Bremen, Gaspard, Meel, 2025]
- Many other problems: Tuple testing, conditional membership, topological sorts [A., Paperman, 2018], perfect matchings, enumerating big results...

Many other directions (talk to me to know more!)

Many generalizations of the **membership problem** of words to regular languages:

- Partial and probabilistic words
  - Many open cases in **data complexity** for CFGs, and in **combined complexity**
- Dynamic membership
  - What about **other updates** (insert/delete, cut and paste...)
  - What about **trees**? **graphs**?
- Enumeration
  - What about **incremental maintenance** of enumeration structures?
    - → [A., Bourhis, Mengel, Niewerth, 2019b], [A., Dziadek, Segoufin, 2025]
- RPQs on graphs
  - What about **simple paths**/**trails**? cf [Bagan et al., 2020] [Martens et al., 2023]
  - What about **RPQs on probabilistic graphs**? [A., van Bremen, Gaspard, Meel, 2025]
- Many other problems: Tuple testing, conditional membership, **topological sorts** [A., Paperman, 2018], **perfect matchings**, enumerating big results...

Thanks for your attention!

References

Aaronson, S., Grier, D., and Schaeffer, L. (2019). A quantum query complexity trichotomy for regular languages. In *FOCS*.

Amarilli, A., Barloy, C., Jachiet, L., and Paperman, C. (2025a). Dynamic membership for regular tree languages. In *MFCS*.

Amarilli, A., Bourhis, P., Mengel, S., and Niewerth, M. (2019a). Constant-Delay Enumeration for Nondeterministic Document Spanners. In *ICDT*.

Amarilli, A., Bourhis, P., Mengel, S., and Niewerth, M. (2019b). Enumeration on trees with tractable combined complexity and efficient updates. In *PODS*.

Amarilli, A., Dziadek, S., and Segoufin, L. (2025b). Constant-time dynamic enumeration of word infixes in a regular language.

Amarilli, A., Gatterbauer, W., Makhija, N., and Monet, M. (2025c). Resilience for regular path queries: Towards a complexity classification. In *PODS*.

Amarilli, A., Groz, B., and Wein, N. (2025d). Edge-minimum walk of modular length in polynomial time. In *ITCS*.

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*.

References iii

Amarilli, A., Manea, F., Ringleb, T., and Schmid, M. L. (2025e). Linear time subsequence and supersequence regex matching. In *MFCS*.

Amarilli, A., Monet, M., Raphaël, P., and Salvati, S. (2025f). On the complexity of language membership for probabilistic words. Under review.

Amarilli, A. and Paperman, C. (2018). Topological sorting under regular constraints. In *ICALP*.

Amarilli, A., van Bremen, T., Gaspard, O., and Meel, K. S. (2025g). Approximating queries on probabilistic graphs. *LMCS*. To appear.

Arenas, M., Croquevielle, L. A., Jayaram, R., and Riveros, C. (2021). #NFA admits an FPRAS: Efficient enumeration, counting, and uniform generation for logspace classes. *Journal of the ACM*, 68(6).

Backurs, A. and Indyk, P. (2016). Which regular expression patterns are hard to match? In *FOCS*.

Bagan, G., Bonifati, A., and Groz, B. (2020). A trichotomy for regular simple path queries on graphs. *Journal of Computer and System Sciences*, 108.

Bodirsky, M., Semanišinová, Ž., and Lutz, C. (2024). The complexity of resilience problems via valued constraint satisfaction problems. In *LICS*.

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

Ganardi, M., Hucke, D., Lohrey, M., Mamouras, K., and Starikovskaya, T. (2025). Regular languages in the sliding window model. *TheoretiCS*, 4.

References v

Ganardi, M., Jachiet, L., Lohrey, M., and Schwentick, T. (2022). Low-latency sliding window algorithms for formal languages. In *IARCS*.

Martens, W., Niewerth, M., and Popp, T. (2023). A trichotomy for regular trail queries. *LMCS*, 19.

Skovbjerg Frandsen, G., Miltersen, P. B., and Skyum, S. (1997). Dynamic word problems. *JACM*, 44(2).

Stearns, R. E. and Hunt III, H. B. (1985). On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM Journal on Computing*, 14(3).