

# Constant-Delay Enumeration for Nondeterministic Document Spanners

---

Antoine Amarilli<sup>1</sup>, Pierre Bourhis<sup>2</sup>, Stefan Mengel<sup>3</sup>, **Matthias Niewerth**<sup>4</sup>

March 27th, 2019

<sup>1</sup>Télécom ParisTech

<sup>2</sup>CNRS CRISTAL

<sup>3</sup>CNRS CRIL

<sup>4</sup>Universität Bayreuth

# Problem: Finding Patterns in Text

- We have a **long text**  $T$ :

```
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 of computer science (office C201-4) in the DIG team of
Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science
awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure.
More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

# Problem: Finding Patterns in Text

- We have a **long text**  $T$ :

```
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 of computer science (office C201-4) in the DIG team of
Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science
awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure.
More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- We want to find a **pattern**  $P$  in the text  $T$ :  
→ Example: find **email addresses**

# Problem: Finding Patterns in Text

- We have a **long text**  $T$ :

```
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 of computer science (office C201-4) in the DIG team of
Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science
awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure.
More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- We want to find a **pattern**  $P$  in the text  $T$ :

→ Example: find **email addresses**

- Write the pattern as a **regular expression**:

$$P := \_ [a-z0-9.]* @ [a-z0-9.]* \_$$

# Problem: Finding Patterns in Text

- We have a **long text**  $T$ :

```
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 of computer science (office C201-4) in the DIG team of
Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science
awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure.
More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- We want to find a **pattern**  $P$  in the text  $T$ :

→ Example: find **email addresses**

- Write the pattern as a **regular expression**:

$$P := \_ [a-z0-9.]* @ [a-z0-9.]* \_$$

→ **How to find the pattern  $P$  efficiently in the text  $T$ ?**

## Solution: Automata

- Convert the **regular expression**  $P$  to an **automaton**  $A$

## Solution: Automata

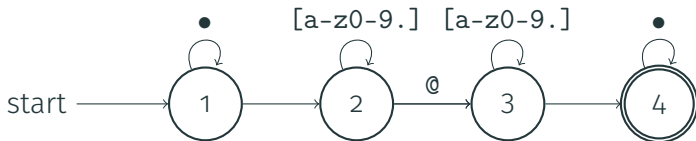
- Convert the **regular expression**  $P$  to an **automaton**  $A$

$$P := \sqcup [a-z0-9.]* @ [a-z0-9.]* \sqcup$$

## Solution: Automata

- Convert the **regular expression**  $P$  to an **automaton**  $A$

$$P := \sqcup [a-z0-9.]* @ [a-z0-9.]* \sqcup$$

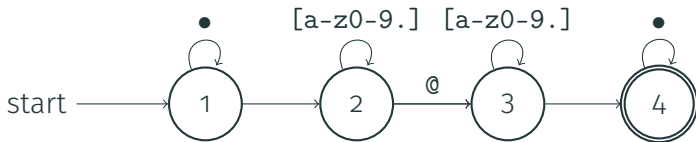




## Solution: Automata

- Convert the **regular expression**  $P$  to an **automaton**  $A$

$$P := \sqcup [a-z0-9.]* @ [a-z0-9.]* \sqcup$$

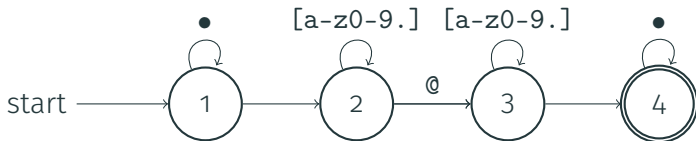


- Then, evaluate the automaton on the **text**  $T$

## Solution: Automata

- Convert the **regular expression**  $P$  to an **automaton**  $A$

$$P := \sqcup [a-z0-9.]* @ [a-z0-9.]* \sqcup$$



- Then, evaluate the automaton on the **text**  $T$

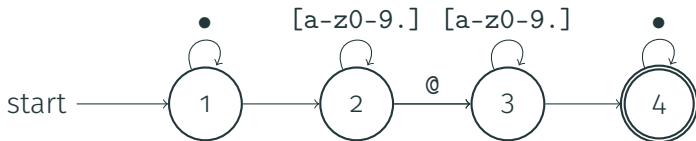
`E m a i l _ a 3 n m @ a 3 n m . n e t _ A f f i l i a t i o n`

- The **complexity** is  $O(|A| \times |T|)$ , i.e., **linear** in  $T$  and **polynomial** in  $P$

## Solution: Automata

- Convert the **regular expression**  $P$  to an **automaton**  $A$

$$P := \sqcup [a-z0-9.]* @ [a-z0-9.]* \sqcup$$



- Then, evaluate the automaton on the **text**  $T$

`E m a i l _ a 3 n m @ a 3 n m . n e t _ A f f i l i a t i o n`

- The **complexity** is  $O(|A| \times |T|)$ , i.e., **linear** in  $T$  and **polynomial** in  $P$   
→ This is **very efficient** in  $T$  and **reasonably efficient** in  $P$

## Actual Problem: Extracting all Patterns

- This only tests **if** the pattern **occurs in** the text!  
→ “YES”

## Actual Problem: Extracting all Patterns

- This only tests **if** the pattern **occurs in** the text!  
→ “YES”
- Goal: find all **substrings** in the text  $T$  which match the pattern  $P$

# Actual Problem: Extracting all Patterns

- This only tests **if** the pattern **occurs in** the text!  
→ “YES”
- Goal: find all **substrings** in the text  $T$  which match the pattern  $P$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
E	m	a	i	l		a	3	n	m	@	a	3	n	m	.	n	e	t		A	f	f	i	l	i	a	t	i	o	n

## Actual Problem: Extracting all Patterns

- This only tests **if** the pattern **occurs in** the text!  
→ “YES”
- Goal: find all **substrings** in the text  $T$  which match the pattern  $P$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30  
E m a i l \_ a 3 n m @ a 3 n m . n e t \_ A f f i l i a t i o n

→ **One match:**  $[5, 20)$

# Formal Problem Statement

- Problem description:



# Formal Problem Statement

- Problem description:
  - Input:
    - A text  $T$

```
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 of computer science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure. test@example.com More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

# Formal Problem Statement

- Problem description:
  - Input:
    - A **text**  $T$

```
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 of computer science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure. test@example.com More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- A **pattern**  $P$  given as a regular expression

$$P := \_ [a-z0-9.]* @ [a-z0-9.]* \_$$

# Formal Problem Statement

- **Problem description:**

- **Input:**

- A **text**  $T$

```
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 of computer science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure. test@example.com More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- A **pattern**  $P$  given as a regular expression

$$P := \sqcup [a-z0-9.]* @ [a-z0-9.]* \sqcup$$

- **Output:** the list of **substrings** of  $T$  that match  $P$ :

[186, 200), [483, 500), ...

# Formal Problem Statement

- **Problem description:**

- **Input:**

- A **text**  $T$

```
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 of computer science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure. test@example.com More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- A **pattern**  $P$  given as a regular expression

$$P := \sqcup [a-z0-9.]* @ [a-z0-9.]* \sqcup$$

- **Output:** the list of **substrings** of  $T$  that match  $P$ :

[186, 200), [483, 500), ...

- **Goal:** be **very efficient** in  $T$  and **reasonably efficient** in  $P$

# Formal Problem Statement

- **Problem description:**

- **Input:**

- A **text**  $T$

```
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 of computer science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure. test@example.com More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- A **sequential document spanner**  $P$  given as a regular expression

$$P := \sqcup x \vdash [a-z0-9.]^* @ [a-z0-9.]^* \dashv x \sqcup$$

- **Output:** the list of **tuples of substrings** of  $T$  that match  $P$ :

[186, 200), [483, 500), ...

- **Goal:** be **very efficient** in  $T$  and **reasonably efficient** in  $P$

# Formal Problem Statement

- **Problem description:**

- **Input:**

- A **text**  $T$

```
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 of computer science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure. test@example.com More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- A **pattern**  $P$  given as a regular expression

$$P := \sqcup [a-z0-9.]* @ [a-z0-9.]* \sqcup$$

- **Output:** the list of **substrings** of  $T$  that match  $P$ :

[186,200), [483,500), ...

- **Goal:** be **very efficient** in  $T$  and **reasonably efficient** in  $P$

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1 o 1

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

[> 1 o 1



# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

[ 1 > o 1

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

[ 1 o > 1

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

[ 1 o 1 ]

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1 [ ] o 1

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1 [ o ] 1

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1 [ o 1 ]

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1 o [ ] 1

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1 o [ 1 ]



# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1 o 1  $\rangle$

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1   o   1

→ **Complexity** is  $O(|T|^2 \times |A| \times |T|)$

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1   o   1

- **Complexity** is  $O(|T|^2 \times |A| \times |T|)$
- Can be **optimized** to  $O(|T|^2 \times |A|)$

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1	o	1
---	---	---

→ **Complexity** is  $O(|T|^2 \times |A| \times |T|)$

→ Can be **optimized** to  $O(|T|^2 \times |A|)$

- **Problem:** We may need to output  $\Omega(|T|^2)$  matching substrings:

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1   o   1

→ **Complexity** is  $O(|T|^2 \times |A| \times |T|)$

→ Can be **optimized** to  $O(|T|^2 \times |A|)$

- **Problem:** We may need to output  $\Omega(|T|^2)$  matching substrings:

- Consider the **text**  $T$ :

aa

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1   o   1

- **Complexity** is  $O(|T|^2 \times |A| \times |T|)$
- Can be **optimized** to  $O(|T|^2 \times |A|)$

- **Problem:** We may need to output  $\Omega(|T|^2)$  matching substrings:

- Consider the **text**  $T$ :

aa

- Consider the **pattern**  $P := a^*$

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1   o   1

→ **Complexity** is  $O(|T|^2 \times |A| \times |T|)$

→ Can be **optimized** to  $O(|T|^2 \times |A|)$

- **Problem:** We may need to output  $\Omega(|T|^2)$  matching substrings:

- Consider the **text**  $T$ :

aa

- Consider the **pattern**  $P := a^*$

- The **number of matches** is  $\Omega(|T|^2)$

# Measuring the Complexity

- **Naive algorithm:** Run the automaton  $A$  on **each substring** of  $T$

1   o   1

- **Complexity** is  $O(|T|^2 \times |A| \times |T|)$
- Can be **optimized** to  $O(|T|^2 \times |A|)$

- **Problem:** We may need to output  $\Omega(|T|^2)$  matching substrings:

- Consider the **text**  $T$ :

aa

- Consider the **pattern**  $P := a^*$
- The **number of matches** is  $\Omega(|T|^2)$

→ We need a **different way** to measure complexity



# Enumeration Algorithms

**Idea:** In real life, we do not want to compute **all the matches** we just need to be able to **enumerate** matches quickly

# Enumeration Algorithms

**Idea:** In real life, we do not want to compute **all the matches** we just need to be able to **enumerate** matches quickly

Q how to find patterns

Search

# Enumeration Algorithms

**Idea:** In real life, we do not want to compute **all the matches** we just need to be able to **enumerate** matches quickly

Results **1 - 20** of **10,514**

# Enumeration Algorithms

**Idea:** In real life, we do not want to compute **all the matches** we just need to be able to **enumerate** matches quickly

Results **1 - 20** of **10,514**

...

# Enumeration Algorithms

**Idea:** In real life, we do not want to compute **all the matches** we just need to be able to **enumerate** matches quickly

Results **1 - 20** of **10,514**

...

View (previous 20 | [next 20](#)) ([20](#) | [50](#) | [100](#) | [250](#) | [500](#))

# Enumeration Algorithms

**Idea:** In real life, we do not want to compute **all the matches** we just need to be able to **enumerate** matches quickly

Results **1 - 20** of **10,514**

...

View (previous 20 | [next 20](#)) ([20](#) | [50](#) | [100](#) | [250](#) | [500](#))

→ Formalization: **enumeration algorithms**

# Formalizing Enumeration Algorithms

```
Antoine Amarilli: Description Name Antoine  
Amarilli. Handle: a3m. Identity Born  
1990-02-07. French national. Appearance as  
of 2017. Auth OpenPGP. OpenId. Bitcoin.  
Contact Email and XMPP a3m@a3m.net  
Affiliation Associate professor ...
```

## Text $T$

□ [a-z0-9.]\*@

[a-z0-9.]\* □

## Pattern $P$

# Formalizing Enumeration Algorithms

```
Antoine Amarilli: Description Name Antoine  
Amarilli. Handle: a3m. Identity Born  
1990-02-07. French national. Appearance as  
of 2017. Auth OpenPGP. OpenId. Bitcoin.  
Contact Email and XMPP a3m@a3m.net  
Affiliation Associate professor ...
```

Text  $T$

```
[a-z0-9.]*@  
[a-z0-9.]*
```

Pattern  $P$

Phase 1:  
Preprocessing





# Formalizing Enumeration Algorithms

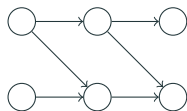
```
Antoine Amarilli: Description Name Antoine  
Amarilli. Handle: a3m. Identity Born  
1990-02-07. French national. Appearance as  
of 2017. Auth OpenPGP. OpenId. Bitcoin.  
Contact Email and XMPP a3m@a3m.net  
Affiliation Associate professor ...
```

Text  $T$

```
□ [a-z0-9.]*@  
[a-z0-9.]* □
```

Pattern  $P$

Phase 1:  
Preprocessing



Index structure

# Formalizing Enumeration Algorithms

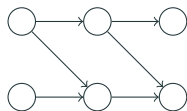
Antoine Amarilli: Description Name Antoine  
Amarilli. Handle: a3m. Identity Born  
1990-02-07. French national. Appearance as  
of 2017. Auth OpenPGP. OpenId. Bitcoin.  
Contact Email and XMPP a3m@a3m.net  
Affiliation Associate professor ...

Text  $T$

$\sqcup [a-z0-9.]*@$   
 $[a-z0-9.]* \sqcup$

Pattern  $P$

Phase 1:  
Preprocessing



Index structure

Phase 2:  
Enumeration

# Formalizing Enumeration Algorithms

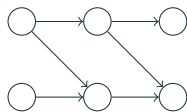
Antoine Amarilli: Description Name Antoine Amarilli. Handle: a3m. Identity Born 1990-02-07. French national. Appearance as of 2017. Auth OpenPGP. OpenId. Bitcoin. Contact Email and XMPP a3m@a3m.net Affiliation Associate professor ...

Text  $T$

$\sqcup [a-z0-9.]*@$   
 $[a-z0-9.]* \sqcup$

Pattern  $P$

Phase 1:  
Preprocessing



Index structure

Phase 2:  
Enumeration

$\{[42, 57]\}$ ,

Results

# Formalizing Enumeration Algorithms

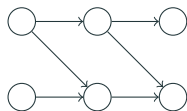
Antoine Amarilli: Description Name Antoine Amarilli. Handle: a3m. Identity Born 1990-02-07. French national. Appearance as of 2017. Auth OpenPGP. OpenId. Bitcoin. Contact Email and XMPP a3m@a3m.net Affiliation Associate professor ...

Text  $T$

$\sqcup [a-z0-9.]*@$   
 $[a-z0-9.]* \sqcup$

Pattern  $P$

Phase 1:  
Preprocessing



Index structure

Phase 2:  
Enumeration

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

Results

# Formalizing Enumeration Algorithms

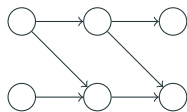
Antoine Amarilli: Description Name Antoine  
Amarilli. Handle: a3m. Identity Born  
1990-02-07. French national. Appearance as  
of 2017. Auth OpenPGP. OpenId. Bitcoin.  
Contact Email and XMPP a3m@a3m.net  
Affiliation Associate professor ...

Text  $T$

$\sqcup [a-z0-9.]*@$   
 $[a-z0-9.]* \sqcup$

Pattern  $P$

Phase 1:  
Preprocessing



Index structure

Phase 2:  
Enumeration

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

Results

Two performance criteria:

- **Total time** for phase 1
  - **Delay between two results** in phase 2
- ... as a function of the **text** and **pattern**

# Complexity of Enumeration Algorithms

- Recall the **inputs** to our problem:
  - A **text**  $T$

```
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 of computer science (office C201-4) in the DIG team of
Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science
awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure.
More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

# Complexity of Enumeration Algorithms

- Recall the **inputs** to our problem:
  - A **text**  $T$

```
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 of computer science (office C201-4) in the DIG team of
Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science
awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure.
More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- A **pattern**  $P$  given as a regular expression

$$P := \square [a-z0-9.]* @ [a-z0-9.]* \square$$

# Complexity of Enumeration Algorithms

- Recall the **inputs** to our problem:

- A **text**  $T$

```
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 of computer science (office C201-4) in the DIG team of
Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science
awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure.
More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- A **pattern**  $P$  given as a regular expression

$$P := \sqcup [a-z0-9]^* @ [a-z0-9]^* \sqcup$$

- What is the **delay** of the **naive algorithm**?



# Complexity of Enumeration Algorithms

- Recall the **inputs** to our problem:

- A **text**  $T$

```
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 of computer science (office C201-4) in the DIG team of
Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science
awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure.
More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- A **pattern**  $P$  given as a regular expression

$$P := \sqcup [a-z0-9]^* @ [a-z0-9]^* \sqcup$$

- What is the **delay** of the **naive algorithm**?

→ it is the **maximal time** to find the next **matching substring**

# Complexity of Enumeration Algorithms

- Recall the **inputs** to our problem:

- A **text**  $T$

```
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 of computer science (office C201-4) in the DIG team of
Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science
awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure.
More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- A **pattern**  $P$  given as a regular expression

$$P := \_ [a-z0-9.]* @ [a-z0-9.]* \_$$

- What is the **delay** of the **naive algorithm**?

→ it is the **maximal time** to find the next **matching substring**

→ i.e.  $O(|T|^2 \times |A|)$ , e.g., if only the **beginning** and **end** match

# Complexity of Enumeration Algorithms

- Recall the **inputs** to our problem:

- A **text**  $T$

```
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 of computer science (office C201-4) in the DIG team of
Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science
awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure.
More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

- A **pattern**  $P$  given as a regular expression

$$P := \_ [a-z0-9.]* @ [a-z0-9.]* \_$$

- What is the **delay** of the **naive algorithm**?

→ it is the **maximal time** to find the next **matching substring**

→ i.e.  $O(|T|^2 \times |A|)$ , e.g., if only the **beginning** and **end** match

→ Can we do **better**?

## Results for Enumerating Pattern Matches

- Existing work has shown the best possible bounds:

# Results for Enumerating Pattern Matches

- Existing work has shown the best possible bounds:

## Theorem [Florenzano et al., 2018]

*We can enumerate all matches of a pattern  $P$  on a text  $T$  with:*

- Preprocessing *linear* in  $T$
- Delay *constant* (independent from  $T$ )

# Results for Enumerating Pattern Matches

- Existing work has shown the best possible bounds **in  $T$** :

## Theorem [Florenzano et al., 2018]

*We can enumerate all matches of a pattern  $P$  on a text  $T$  with:*

- Preprocessing **linear** in  $T$  and **exponential** in  $P$
- Delay **constant** (independent from  $T$ ) and **exponential** in  $P$

→ **Problem:** Only efficient for **deterministic** automata!

# Results for Enumerating Pattern Matches

- Existing work has shown the best possible bounds **in  $T$** :

## Theorem [Florenzano et al., 2018]

*We can enumerate all matches of a pattern  $P$  on a text  $T$  with:*

- Preprocessing **linear** in  $T$  and **exponential in  $P$**
- Delay **constant** (independent from  $T$ ) and **exponential in  $P$**

→ **Problem:** Only efficient for **deterministic** automata!

- **Our contribution** is:

# Results for Enumerating Pattern Matches

- Existing work has shown the best possible bounds **in  $T$** :

## Theorem [Florenzano et al., 2018]

We can enumerate all matches of a pattern  $P$  on a text  $T$  with:

- Preprocessing **linear** in  $T$  and **exponential** in  $P$
- Delay **constant** (independent from  $T$ ) and **exponential** in  $P$

→ **Problem**: Only efficient for **deterministic** automata!

- **Our contribution** is:

## Theorem

We can enumerate all matches of a pattern  $P$  on a text  $T$  with:

- Preprocessing in  $O(|T| \times \text{Poly}(P))$
- Delay **polynomial** in  $P$  and **independent** from  $T$



# Automaton Formalism

- We use automata that read letters and **capture variables**

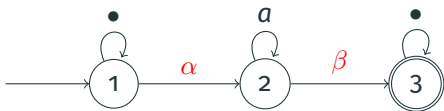
# Automaton Formalism

- We use automata that read letters and **capture variables**  
→ **Example:**  $P := \bullet^* \alpha a^* \beta \bullet^*$

# Automaton Formalism

- We use automata that read letters and **capture variables**

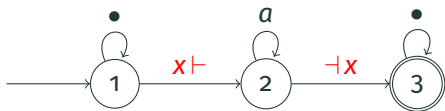
→ **Example:**  $P := \bullet^* \alpha a^* \beta \bullet^*$



# Automaton Formalism

- We use automata that read letters and **capture variables**

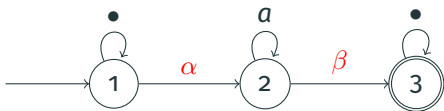
→ **Example:**  $P := \bullet^* x \vdash a^* \neg x \bullet^*$



# Automaton Formalism

- We use automata that read letters and **capture variables**

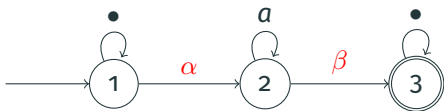
→ **Example:**  $P := \bullet^* \alpha a^* \beta \bullet^*$



# Automaton Formalism

- We use automata that read letters and **capture variables**

→ **Example:**  $P := \bullet^* \alpha a^* \beta \bullet^*$

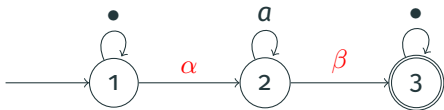


- Semantics of the automaton **A**:
  - Reads** letters from the text
  - Guesses** variables at positions in the text

# Automaton Formalism

- We use automata that read letters and **capture variables**

→ **Example:**  $P := \bullet^* \alpha a^* \beta \bullet^*$

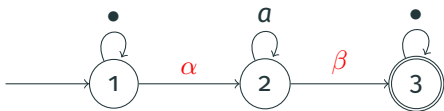


- Semantics of the automaton **A**:
  - Reads** letters from the text
  - Guesses** variables at positions in the text
- **Output:** tuples  $\langle \alpha : i, \beta : j \rangle$  such that **A** has an accepting run reading  $\alpha$  at position  $i$  and  $\beta$  at  $j$

# Automaton Formalism

- We use automata that read letters and **capture variables**

→ **Example:**  $P := \bullet^* \alpha a^* \beta \bullet^*$



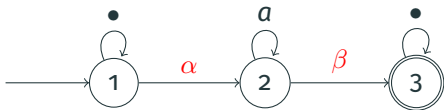
- Semantics of the automaton **A**:
  - Reads** letters from the text
  - Guesses** variables at positions in the text
- **Output:** tuples  $\langle \alpha : i, \beta : j \rangle$  such that **A** has an accepting run reading  $\alpha$  at position  $i$  and  $\beta$  at  $j$
- Assumption:** There is no run for which **A** reads the same **capture variable** twice at the same **position**



# Automaton Formalism

- We use automata that read letters and **capture variables**

→ **Example:**  $P := \bullet^* \alpha a^* \beta \bullet^*$



- Semantics of the automaton **A**:
  - Reads** letters from the text
  - Guesses** variables at positions in the text
- **Output:** tuples  $\langle \alpha : i, \beta : j \rangle$  such that **A** has an accepting run reading  $\alpha$  at position  $i$  and  $\beta$  at  $j$
- Assumption:** There is no run for which **A** reads the same **capture variable** twice at the same **position**
- Challenge:** Because of **nondeterminism** we can have many different runs of **A** producing the same tuple!

## Proof idea: Product DAG

Compute a **product DAG** of the text  $T$  and of the automaton  $A$

## Proof idea: Product DAG

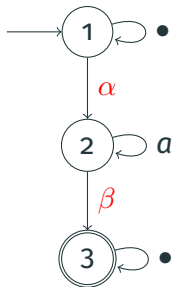
Compute a **product DAG** of the text  $T$  and of the automaton  $A$

**Example:** Text  $T :=$  aaaba and  $P := \bullet^* \alpha a^* \beta \bullet^*$ ,

## Proof idea: Product DAG

Compute a **product DAG** of the text  $T$  and of the automaton  $A$

**Example:** Text  $T := \boxed{\text{aaaba}}$  and  $P := \bullet^* \alpha a^* \beta \bullet^*$ ,

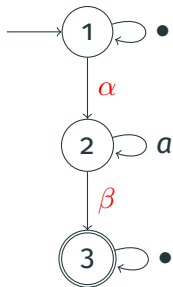


## Proof idea: Product DAG

Compute a **product DAG** of the text  $T$  and of the automaton  $A$

**Example:** Text  $T :=$  aaaba and  $P := \bullet^* \alpha a^* \beta \bullet^*$ ,

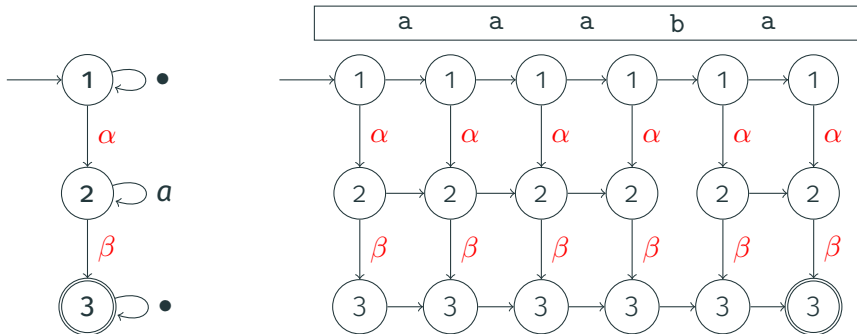
a      a      a      b      a



# Proof idea: Product DAG

Compute a **product DAG** of the text  $T$  and of the automaton  $A$

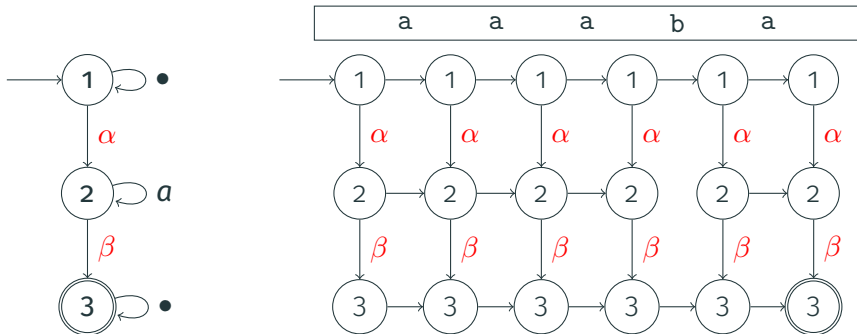
**Example:** Text  $T :=$  aaaba and  $P := \bullet^* \alpha a^* \beta \bullet^*$ ,



# Proof idea: Product DAG

Compute a **product DAG** of the text  $T$  and of the automaton  $A$

**Example:** Text  $T :=$  aaaba and  $P := \bullet^* \alpha a^* \beta \bullet^*$ ,

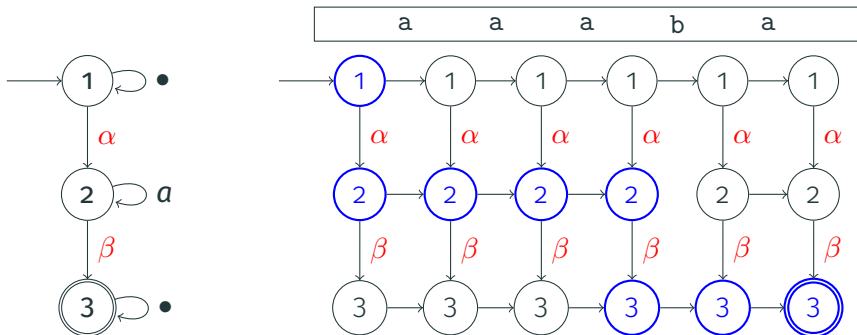


→ Each **path** in the **product DAG** corresponds to a **match**

# Proof idea: Product DAG

Compute a **product DAG** of the text  $T$  and of the automaton  $A$

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



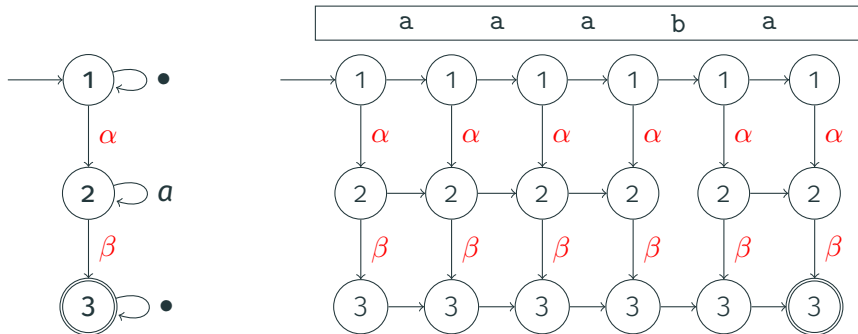
$\rightarrow$  Each **path** in the **product DAG** corresponds to a **match**



# Proof idea: Product DAG

Compute a **product DAG** of the text  $T$  and of the automaton  $A$

**Example:** Text  $T :=$  aaaba and  $P := \bullet^* \alpha a^* \beta \bullet^*$ ,



→ Each **path** in the **product DAG** corresponds to a **match**

→ **Challenge:** Enumerate paths but avoid **duplicate matches** and do not **waste time** to ensure constant delay

# Proof ingredients

Several ingredients to do this efficiently

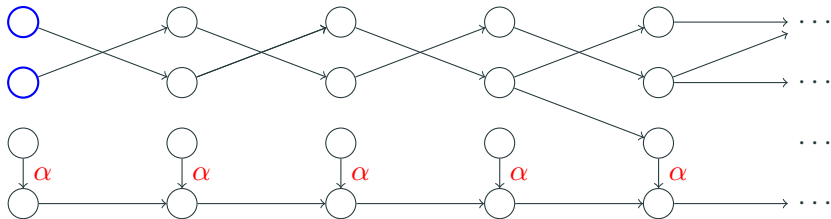
- Prune non-accepting paths
- Use shortcuts (pointers) to skip long paths
- Flashlight search

## Proof ingredient: jump pointers to save time

- **Issue:** When we can't assign variables, we do not make **progress**

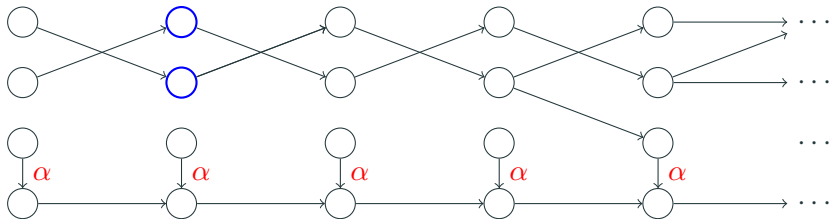
## Proof ingredient: jump pointers to save time

- Issue:** When we can't assign variables, we do not make **progress**



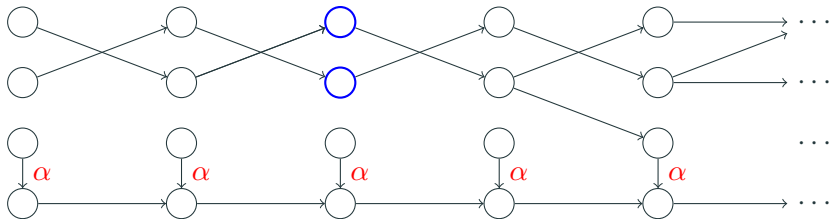
# Proof ingredient: jump pointers to save time

- Issue:** When we can't assign variables, we do not make **progress**



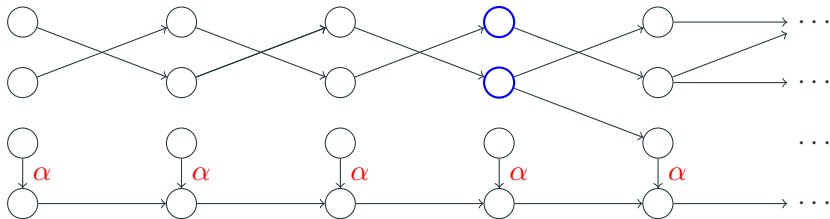
# Proof ingredient: jump pointers to save time

- Issue:** When we can't assign variables, we do not make **progress**



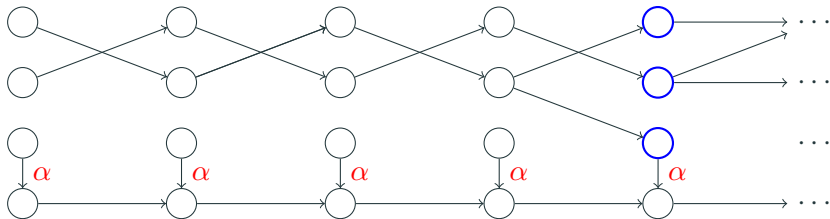
## Proof ingredient: jump pointers to save time

- Issue:** When we can't assign variables, we do not make **progress**



## Proof ingredient: jump pointers to save time

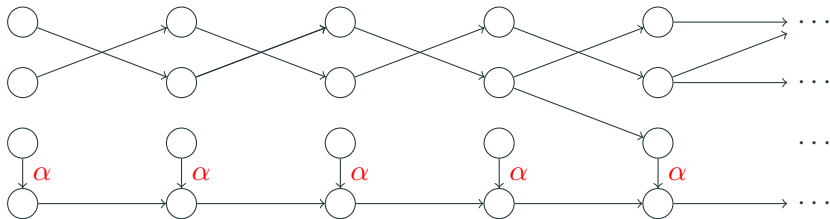
- Issue:** When we can't assign variables, we do not make **progress**





## Proof ingredient: jump pointers to save time

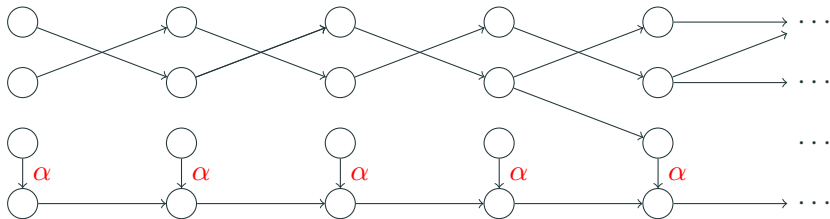
- Issue:** When we can't assign variables, we do not make **progress**



- Idea:** Directly **jump** to the reachable states at the next position where we can assign a variable

# Proof ingredient: jump pointers to save time

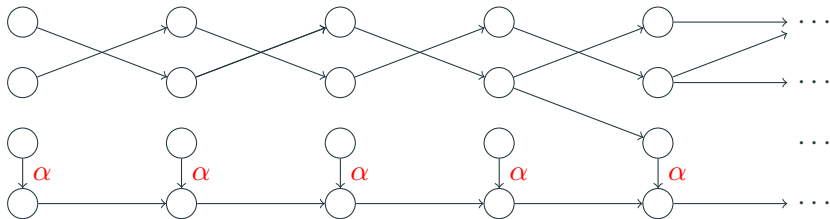
- **Issue:** When we can't assign variables, we do not make **progress**



- **Idea:** Directly **jump** to the reachable states at the next position where we can assign a variable
- **Challenge:** Preprocessing in **linear time** in  $T$  and **polynomial** in  $A$ :

# Proof ingredient: jump pointers to save time

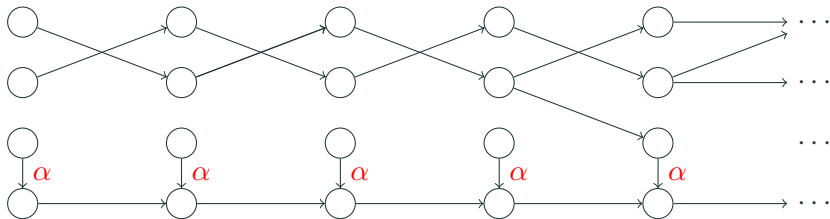
- **Issue:** When we can't assign variables, we do not make **progress**



- **Idea:** Directly **jump** to the reachable states at the next position where we can assign a variable
- **Challenge:** Preprocessing in **linear time** in  $T$  and **polynomial** in  $A$ :
  - Compute for each state the **next position** where we can reach some state that can assign a variable

# Proof ingredient: jump pointers to save time

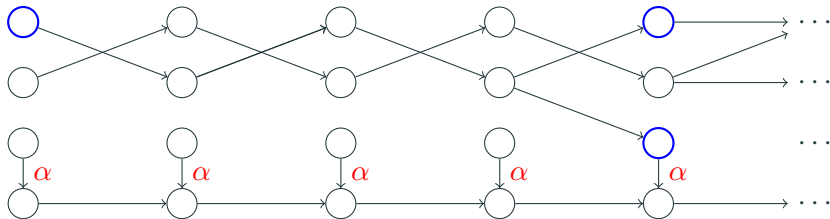
- Issue:** When we can't assign variables, we do not make **progress**



- Idea:** Directly **jump** to the reachable states at the next position where we can assign a variable
- Challenge:** Preprocessing in **linear time** in  $T$  and **polynomial** in  $A$ :
  - Compute for each state the **next position** where we can reach some state that can assign a variable
  - Compute at each position  $i$  the **transitive closure** to all positions  $j$  such that  $j$  is the next position of some state at  $i$  (there are  $\leq |A|^j$ )

# Proof ingredient: jump pointers to save time

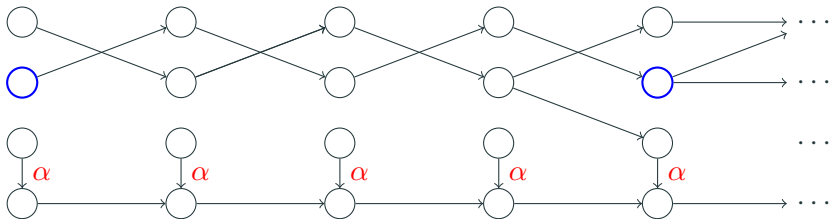
- **Issue:** When we can't assign variables, we do not make **progress**



- **Idea:** Directly **jump** to the reachable states at the next position where we can assign a variable
- **Challenge:** Preprocessing in **linear time** in  $T$  and **polynomial** in  $A$ :
  - Compute for each state the **next position** where we can reach some state that can assign a variable
  - Compute at each position  $i$  the **transitive closure** to all positions  $j$  such that  $j$  is the next position of some state at  $i$  (there are  $\leq |A|^i$ )

# Proof ingredient: jump pointers to save time

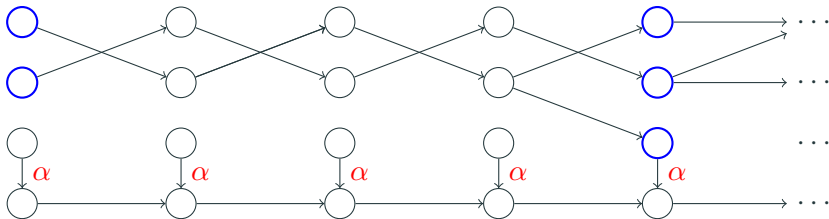
- **Issue:** When we can't assign variables, we do not make **progress**



- **Idea:** Directly **jump** to the reachable states at the next position where we can assign a variable
- **Challenge:** Preprocessing in **linear time** in  $T$  and **polynomial** in  $A$ :
  - Compute for each state the **next position** where we can reach some state that can assign a variable
  - Compute at each position  $i$  the **transitive closure** to all positions  $j$  such that  $j$  is the next position of some state at  $i$  (there are  $\leq |A|$ )

# Proof ingredient: jump pointers to save time

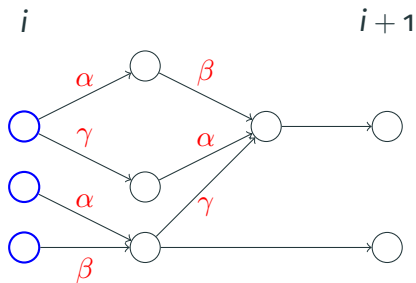
- **Issue:** When we can't assign variables, we do not make **progress**



- **Idea:** Directly **jump** to the reachable states at the next position where we can assign a variable
- **Challenge:** Preprocessing in **linear time** in  $T$  and **polynomial** in  $A$ :
  - Compute for each state the **next position** where we can reach some state that can assign a variable
  - Compute at each position  $i$  the **transitive closure** to all positions  $j$  such that  $j$  is the next position of some state at  $i$  (there are  $\leq |A|$ )

## Proof ingredient: flashlight search

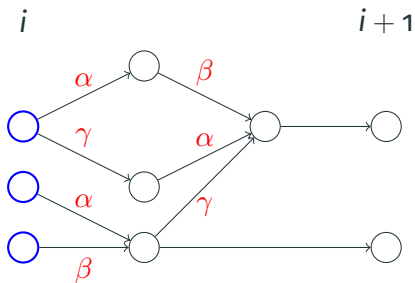
- Issue:** Finding which **variable sets** we can assign at position  $i$ ?





## Proof ingredient: flashlight search

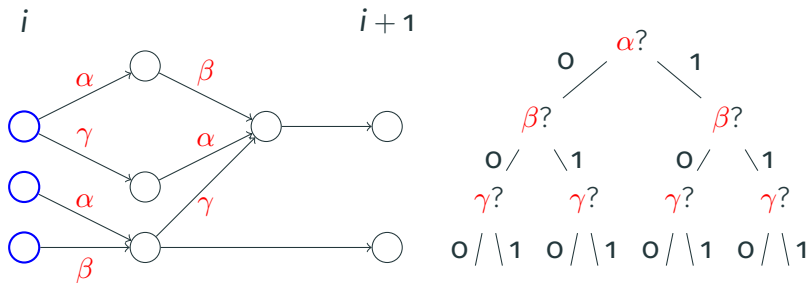
- **Issue:** Finding which **variable sets** we can assign at position  $i$ ?



- **Idea:** Explore a **decision tree** on the variables (built on the fly)

# Proof ingredient: flashlight search

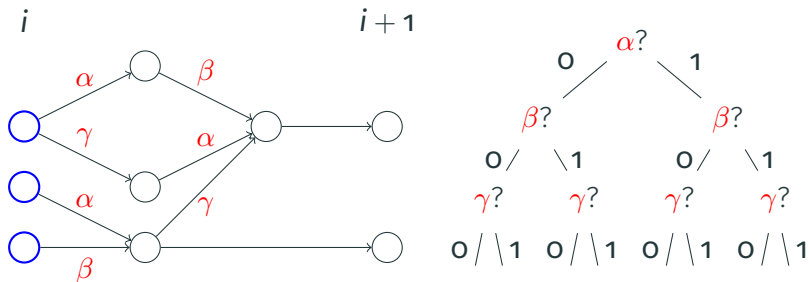
- Issue:** Finding which **variable sets** we can assign at position  $i$ ?



- Idea:** Explore a **decision tree** on the variables (built on the fly)

# Proof ingredient: flashlight search

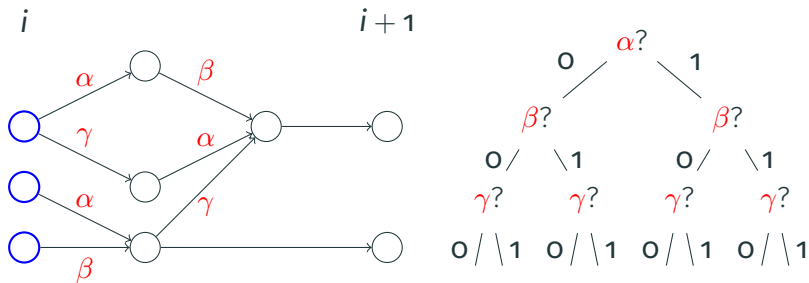
- **Issue:** Finding which **variable sets** we can assign at position  $i$ ?



- **Idea:** Explore a **decision tree** on the variables (built on the fly)
- At each decision tree **node**, find the reachable **states** which have **all required variables** (1) and **no forbidden variables** (0)

# Proof ingredient: flashlight search

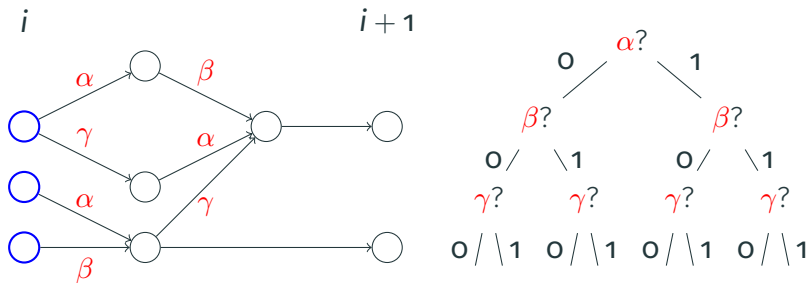
- **Issue:** Finding which **variable sets** we can assign at position  $i$ ?



- **Idea:** Explore a **decision tree** on the variables (built on the fly)
- At each decision tree **node**, find the reachable **states** which have **all required variables** (1) and **no forbidden variables** (0)  
→ **Assumption:** we don't see the same variable **twice** on a path

# Proof ingredient: flashlight search

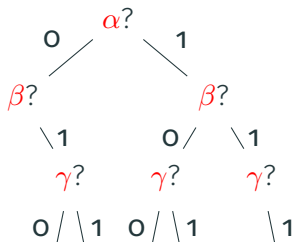
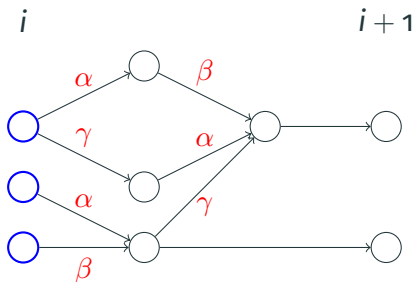
- **Issue:** Finding which **variable sets** we can assign at position  $i$ ?



- **Idea:** Explore a **decision tree** on the variables (built on the fly)
- At each decision tree **node**, find the reachable **states** which have **all required variables** (1) and **no forbidden variables** (0)  
→ **Assumption:** we don't see the same variable **twice** on a path

# Proof ingredient: flashlight search

- Issue:** Finding which **variable sets** we can assign at position  $i$ ?



- Idea:** Explore a **decision tree** on the variables (built on the fly)
- At each decision tree **node**, find the reachable **states** which have **all required variables** (1) and **no forbidden variables** (0)
  - **Assumption:** we don't see the same variable **twice** on a path

## **Summary and Future Work**

---

# Main Result and Future Work

## Theorem

Given a sequential document spanner  $P$  and text  $T$ , we can enumerate with:

Preprocessing  $O(|P|^{\omega+1} \times |T|)$

Delay  $O(|\mathcal{V}^3| \times |P|^2)$

$\mathcal{V}$ : Set of Variables

$\omega$ : Exponent for Boolean matrix multiplication



# Main Result and Future Work

## Theorem

Given a sequential document spanner  $P$  and text  $T$ , we can enumerate with:

Preprocessing  $O(|P|^{\omega+1} \times |T|)$

Delay  $O(|\mathcal{V}^3| \times |P|^2)$

# Main Result and Future Work

## Theorem

Given a sequential document spanner  $P$  and text  $T$ , we can enumerate with:

$$\text{Preprocessing } O(|P|^{\omega+1} \times |T|)$$

$$\text{Delay } O(|\mathcal{V}^3| \times |P|^2)$$

## Extensions and future work:

- Extending the results from text to **trees**

# Main Result and Future Work

## Theorem

Given a sequential document spanner  $P$  and text  $T$ , we can enumerate with:

$$\text{Preprocessing } O(|P|^{\omega+1} \times |T|)$$

$$\text{Delay } O(|\mathcal{V}^3| \times |P|^2)$$

## Extensions and future work:

- Extending the results from text to **trees**
- Supporting **updates** on the input data

# Main Result and Future Work

## Theorem

Given a sequential document spanner  $P$  and text  $T$ , we can enumerate with:

Preprocessing  $O(|P|^{\omega+1} \times |T|)$

Delay  $O(|\mathcal{V}^3| \times |P|^2)$

## Extensions and future work:

- Extending the results from text to **trees**
  - Supporting **updates** on the input data
- } PODS 2019

# Main Result and Future Work

## Theorem

Given a sequential document spanner  $P$  and text  $T$ , we can enumerate with:

Preprocessing  $O(|P|^{\omega+1} \times |T|)$

Delay  $O(|\mathcal{V}^3| \times |P|^2)$

## Extensions and future work:

- Extending the results from text to **trees**
  - Supporting **updates** on the input data
  - Enumerating results in a relevant **order?**
- } PODS 2019

# Main Result and Future Work

## Theorem

Given a sequential document spanner  $P$  and text  $T$ , we can enumerate with:

Preprocessing  $O(|P|^{\omega+1} \times |T|)$

Delay  $O(|\mathcal{V}^3| \times |P|^2)$

## Extensions and future work:

- Extending the results from text to **trees**
  - Supporting **updates** on the input data
  - Enumerating results in a relevant **order?**
  - Testing how well our methods perform in **practice**
- } PODS 2019

# Main Result and Future Work

## Theorem

Given a sequential document spanner  $P$  and text  $T$ , we can enumerate with:

Preprocessing  $O(|P|^{\omega+1} \times |T|)$

Delay  $O(|\mathcal{V}^3| \times |P|^2)$

## Extensions and future work:

- Extending the results from text to **trees**
  - Supporting **updates** on the input data
  - Enumerating results in a relevant **order?**
  - Testing how well our methods perform in **practice**
- } PODS 2019
- } Rémi Dupré

# Main Result and Future Work

## Theorem

Given a sequential document spanner  $P$  and text  $T$ , we can enumerate with:

Preprocessing  $O(|P|^{\omega+1} \times |T|)$

Delay  $O(|\mathcal{V}^3| \times |P|^2)$

## Extensions and future work:

- Extending the results from text to **trees**
  - Supporting **updates** on the input data
  - Enumerating results in a relevant **order**?
  - Testing how well our methods perform in **practice**
- } PODS 2019
- } Rémi Dupré

Thanks for your attention!



 Florenzano, F., Riveros, C., Ugarte, M., Vansummeren, S., and Vrgoc, D. (2018).

**Constant delay algorithms for regular document spanners.**

In *PODS*.