Enumerating Pattern Matches in Texts and Trees

Antoine Amarilli[1], Pierre Bourhis[2], Stefan Mengel[3], Matthias Niewerth[4]

October 24th, 2019

[1]Télécom Paris

[2]CNRS CRIStAL

[3]CNRS CRIL

[4]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 Paris, 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 Paris, 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 Paris, 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 Paris, 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**:

$$P := \texttt{␣ [a-z0-9.]* @ [a-z0-9.]* ␣}$$

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

Solution: Automata

- Convert the regular expression P to an automaton A

Solution: Automata

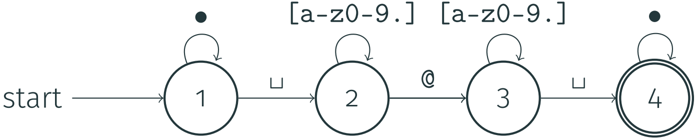
- Convert the regular expression P to an automaton A

$$P := \textvisiblespace\ \texttt{[a-z0-9.]}^{*}\ \texttt{@}\ \texttt{[a-z0-9.]}^{*}\ \textvisiblespace$$

Solution: Automata

- Convert the regular expression P to an automaton A

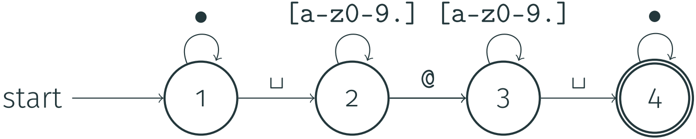
$$P := \texttt{␣ [a-z0-9.]* @ [a-z0-9.]* ␣}$$



Solution: Automata

- Convert the regular expression P to an automaton A

$$P := \texttt{␣ [a-z0-9.]* @ [a-z0-9.]* ␣}$$

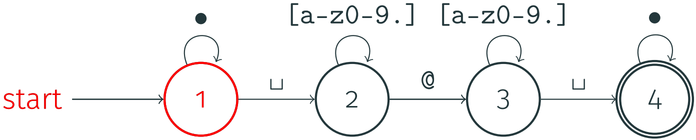


- Then, evaluate the automaton on the text T

Solution: Automata

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

$$P := \textvisiblespace\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \textvisiblespace$$



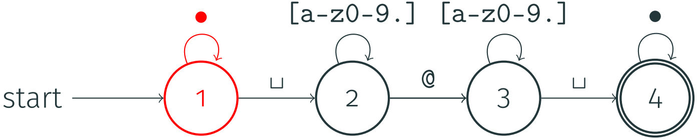
- Then, evaluate the automaton on the **text** *T*

Email␣a3nm@a3nm.net␣Affiliation

Solution: Automata

- Convert the regular expression P to an automaton A

$$P := \texttt{␣ [a-z0-9.]* @ [a-z0-9.]* ␣}$$



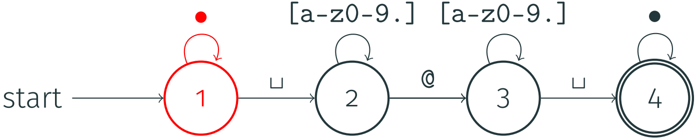
- Then, evaluate the automaton on the text T

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

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

$$P := \textvisiblespace\ \texttt{[a-z0-9.]}^{*}\ \texttt{@}\ \texttt{[a-z0-9.]}^{*}\ \textvisiblespace$$



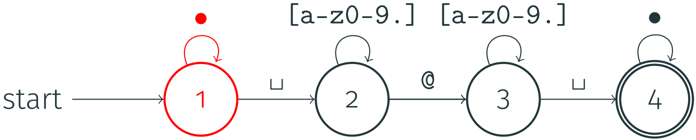
- Then, evaluate the automaton on the text *T*

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

- Convert the regular expression P to an automaton A

$$P := \text{␣ [a-z0-9.]* @ [a-z0-9.]* ␣}$$



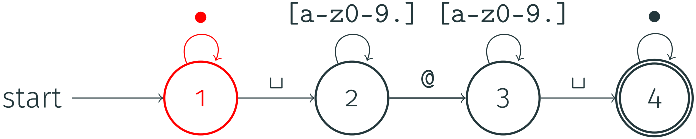
- Then, evaluate the automaton on the text T

```
Email␣a3nm@a3nm.net␣Affiliation
```

Solution: Automata

- Convert the regular expression P to an automaton A

$$P := \texttt{␣ [a-z0-9.]* @ [a-z0-9.]* ␣}$$



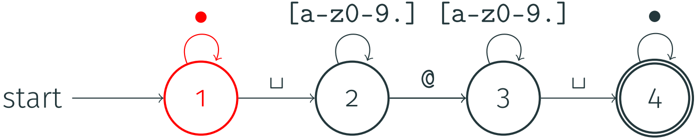
- Then, evaluate the automaton on the text T

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

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

$$P := \sqcup\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \sqcup$$



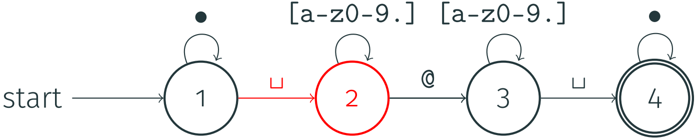
- Then, evaluate the automaton on the **text** *T*

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

- Convert the regular expression P to an automaton A

$$P := \textvisiblespace\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \textvisiblespace$$



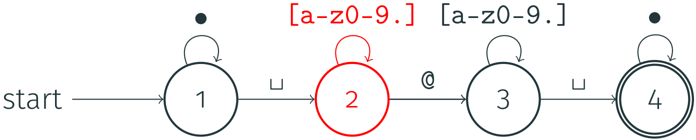
- Then, evaluate the automaton on the text T

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

Solution: Automata

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

$P :=$ `␣ [a-z0-9.]* @ [a-z0-9.]* ␣`



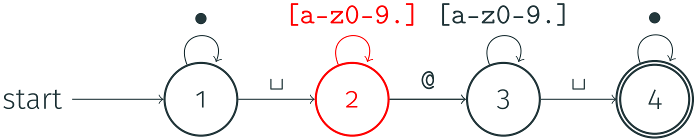
- Then, evaluate the automaton on the **text** *T*

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

- Convert the regular expression P to an automaton A

$$P := \texttt{␣ [a-z0-9.]}^* \texttt{ @ [a-z0-9.]}^* \texttt{ ␣}$$



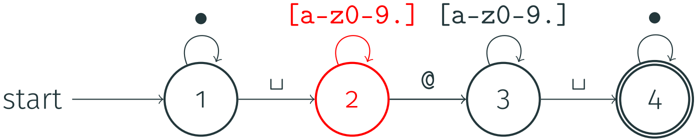
- Then, evaluate the automaton on the text T

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

- Convert the regular expression P to an automaton A

$$P := \textvisiblespace\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \textvisiblespace$$



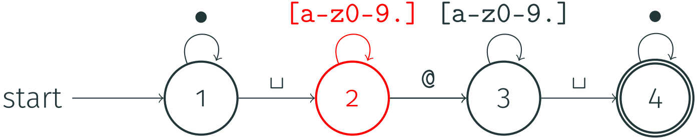
- Then, evaluate the automaton on the text T

Email␣a3nm@a3nm.net␣Affiliation

Solution: Automata

- Convert the regular expression P to an automaton A

$P :=$ ␣ [a-z0-9.]* @ [a-z0-9.]* ␣



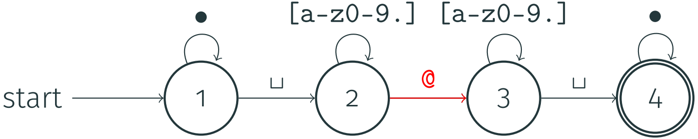
- Then, evaluate the automaton on the text T

Email␣a3nm@a3nm.net␣Affiliation

Solution: Automata

- Convert the regular expression P to an automaton A

$P :=$ `␣ [a-z0-9.]* @ [a-z0-9.]* ␣`



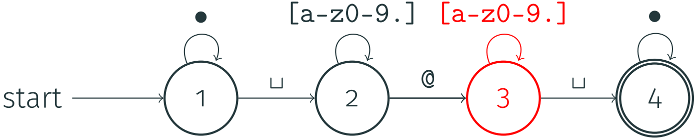
- Then, evaluate the automaton on the text T

```
Email␣a3nm@a3nm.net␣Affiliation
```

Solution: Automata

- Convert the regular expression P to an automaton A

$P :=$ `␣ [a-z0-9.]* @ [a-z0-9.]* ␣`



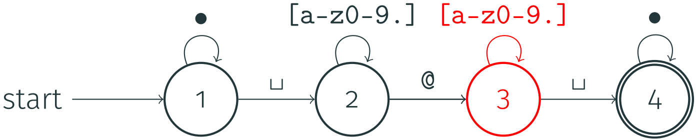
- Then, evaluate the automaton on the text T

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

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

$$P := \texttt{␣ [a-z0-9.]}^{*}\ \texttt{@ [a-z0-9.]}^{*}\ \texttt{␣}$$



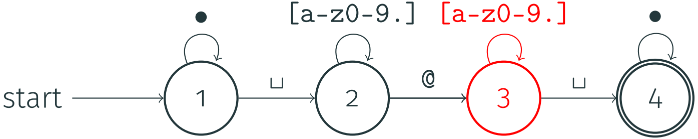
- Then, evaluate the automaton on the text *T*

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

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

$P :=$ ␣ [a-z0-9.]* @ [a-z0-9.]* ␣



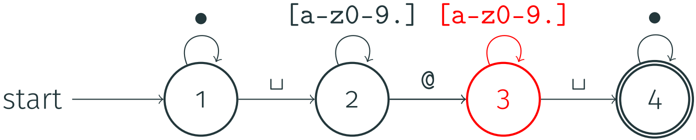
- Then, evaluate the automaton on the **text** *T*

Email␣a3nm@a3nm.net␣Affiliation

Solution: Automata

- Convert the regular expression P to an automaton A

$$P := \texttt{␣ [a-z0-9.]}^* \texttt{ @ [a-z0-9.]}^* \texttt{ ␣}$$



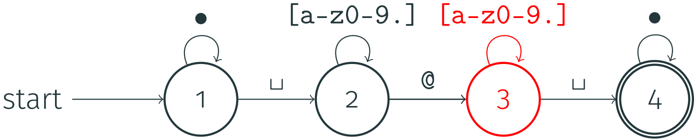
- Then, evaluate the automaton on the text T

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

- Convert the regular expression P to an automaton A

$$P := \textvisiblespace\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \textvisiblespace$$



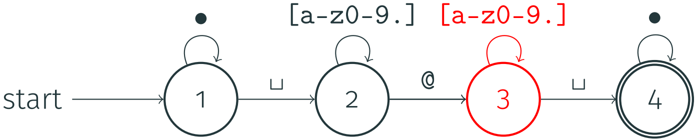
- Then, evaluate the automaton on the text T

Email␣a3nm@a3nm.net␣Affiliation

Solution: Automata

- Convert the regular expression P to an automaton A

$P :=$ ␣ [a-z0-9.]* @ [a-z0-9.]* ␣



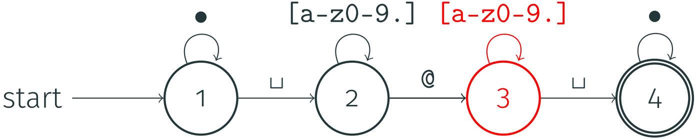
- Then, evaluate the automaton on the text T

Email␣a3nm@a3nm.net␣Affiliation

Solution: Automata

- Convert the regular expression P to an automaton A

$P :=$ `␣ [a-z0-9.]* @ [a-z0-9.]* ␣`



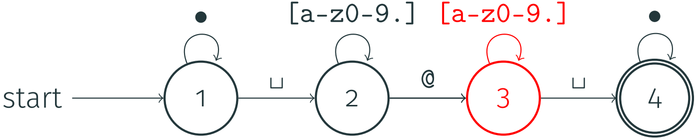
- Then, evaluate the automaton on the text T

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

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

$$P := \sqcup\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \sqcup$$



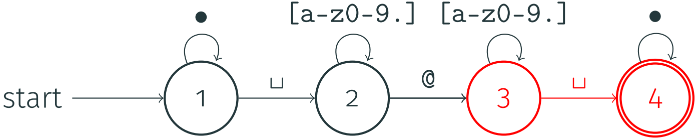
- Then, evaluate the automaton on the text *T*

Email␣a3nm@a3nm.net␣Affiliation

Solution: Automata

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

$P :=$ ␣ [a-z0-9.]* @ [a-z0-9.]* ␣



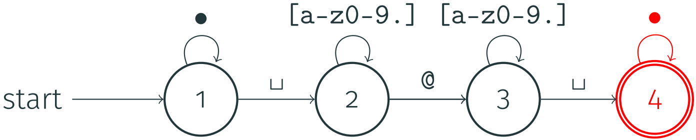
- Then, evaluate the automaton on the text *T*

Email␣a3nm@a3nm.net␣Affiliation

Solution: Automata

- Convert the regular expression P to an automaton A

$P :=$ `␣ [a-z0-9.]* @ [a-z0-9.]* ␣`



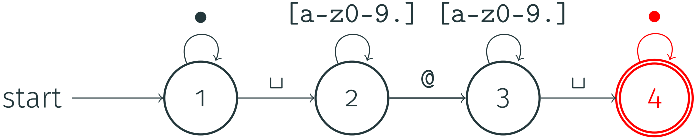
- Then, evaluate the automaton on the text T

```
Email␣a3nm@a3nm.net␣Affiliation
```

Solution: Automata

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

$$P := \sqcup\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \sqcup$$



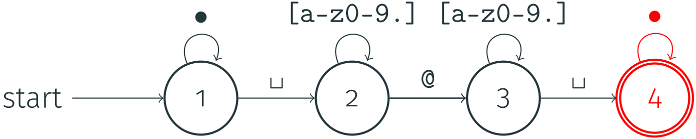
- Then, evaluate the automaton on the text *T*

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

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

$P :=$ ␣ [a-z0-9.]* @ [a-z0-9.]* ␣



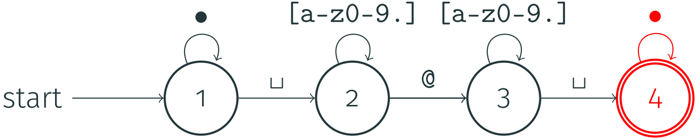
- Then, evaluate the automaton on the text *T*

Email␣a3nm@a3nm.net␣Affiliation

Solution: Automata

- Convert the regular expression P to an automaton A

$P :=$ `␣ [a-z0-9.]* @ [a-z0-9.]* ␣`



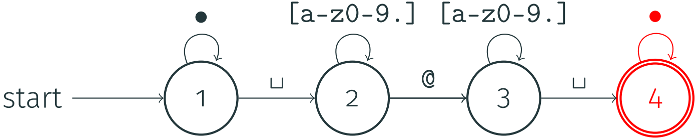
- Then, evaluate the automaton on the text T

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

- Convert the regular expression P to an automaton A

$$P := \text{␣ [a-z0-9.]* @ [a-z0-9.]* ␣}$$



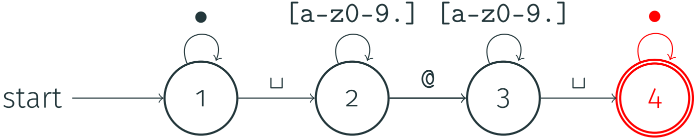
- Then, evaluate the automaton on the text T

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

- Convert the regular expression P to an automaton A

$$P := \texttt{\textvisiblespace\ [a-z0-9.]}^{*}\ \texttt{@ [a-z0-9.]}^{*}\ \texttt{\textvisiblespace}$$



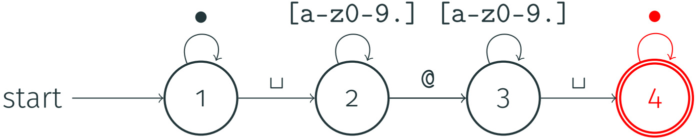
- Then, evaluate the automaton on the text T

Email␣a3nm@a3nm.net␣Affiliation

Solution: Automata

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

$$P :=\ \texttt{␣ [a-z0-9.]* @ [a-z0-9.]* ␣}$$



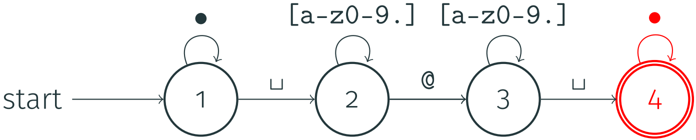
- Then, evaluate the automaton on the **text** *T*

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

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

$P :=$ `␣ [a-z0-9.]* @ [a-z0-9.]* ␣`



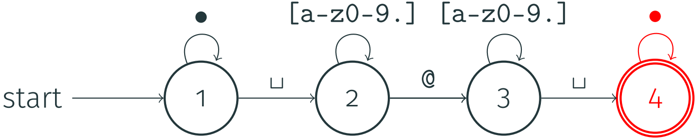
- Then, evaluate the automaton on the text *T*

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

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

$P :=$ `␣ [a-z0-9.]* @ [a-z0-9.]* ␣`



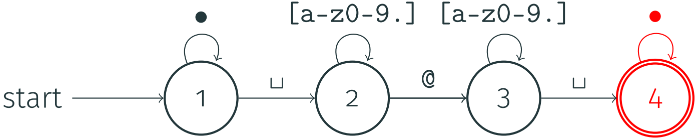
- Then, evaluate the automaton on the **text** *T*

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

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

$P :=$ `␣ [a-z0-9.]* @ [a-z0-9.]* ␣`



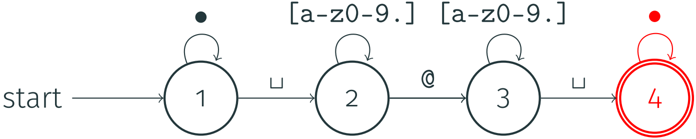
- Then, evaluate the automaton on the text *T*

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

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

$$P := \text{␣ [a-z0-9.]* @ [a-z0-9.]* ␣}$$



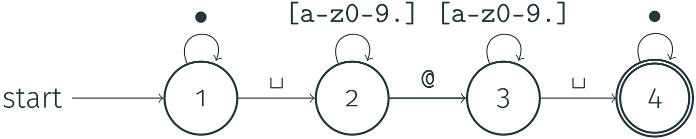
- Then, evaluate the automaton on the **text** T

`Email␣a3nm@a3nm.net␣Affiliation`

Solution: Automata

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

$P :=$ `␣ [a-z0-9.]* @ [a-z0-9.]* ␣`



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

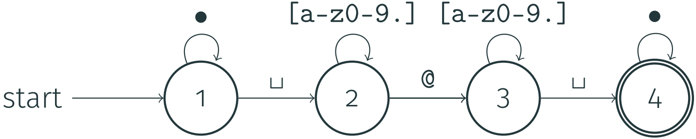
`Email␣a3nm@a3nm.net␣Affiliation`

- 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

$$P := \sqcup\ \texttt{[a-z0-9.]}^{*}\ \texttt{@}\ \texttt{[a-z0-9.]}^{*}\ \sqcup$$



- Then, evaluate the automaton on the text T

Email␣a3nm@a3nm.net␣Affiliation

- 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!

  $\rightarrow$ ‘‘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:** $[5, 20\rangle$

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:** $[5, 20\rangle$

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 Paris,
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 Paris,
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

$$P := \textvisiblespace\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \textvisiblespace$$

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 Paris,
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

$$P := \sqcup\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \sqcup$$

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

$$[186, 200\rangle, \quad [483, 500\rangle, \quad \ldots$$

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 Paris,
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

$$P := \textvisiblespace\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \textvisiblespace$$

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

$$[186, 200\rangle, \quad [483, 500\rangle, \quad \ldots$$

- 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*

```
    l   o   l
```

Measuring the Complexity

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

```
[⟩ l   o   l
```

Measuring the Complexity

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

$[$ l $\rangle$ o   l

Measuring the Complexity

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

`[ l  o ⟩ l`

Measuring the Complexity

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

```
[ l   o   l ⟩
```

Measuring the Complexity

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

```
    l [⟩ o   l
```

Measuring the Complexity

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

```
    l [ o ⟩ l
```

Measuring the Complexity

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

```
    l [ o   l ⟩
```

Measuring the Complexity

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

```
    l   o [⟩ l
```

Measuring the Complexity

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

```
    l   o [ l ⟩
```

Measuring the Complexity

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

```
    l   o   l [⟩
```

Measuring the Complexity

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

```
    l   o   l
```

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

Measuring the Complexity

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

```
    l   o   l
```

→ **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

```
    l   o   l
```

  → 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

```
    l    o    l
```

→ **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:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Measuring the Complexity

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

```
    l   o   l
```

  → **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:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

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

Measuring the Complexity

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

  `l o l`

  → **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:

    `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`

  - 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

```
    l    o    l
```

→ **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:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

  - 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

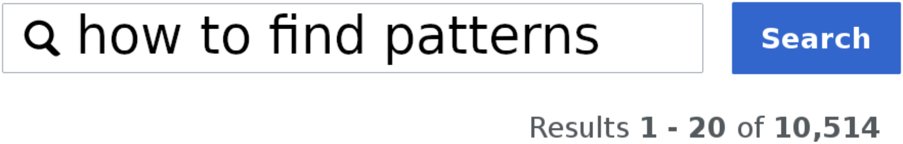
Enumeration Algorithms

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

how to find patterns

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



how to find patterns

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

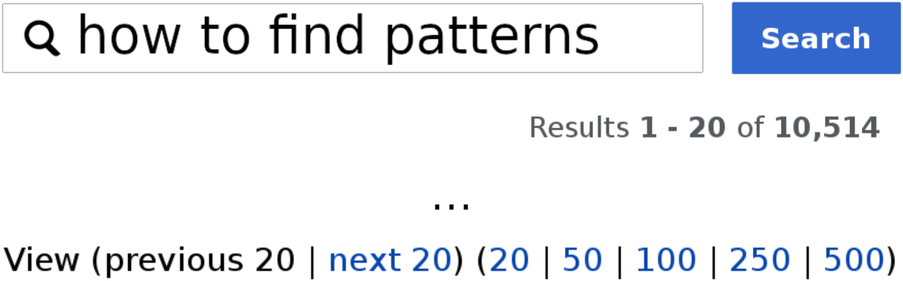
how to find patterns

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



how to find patterns

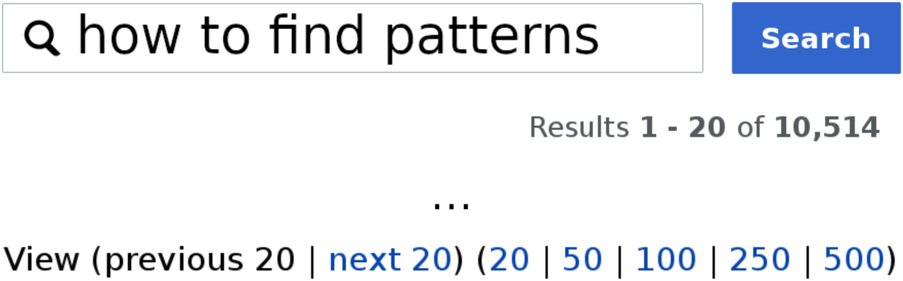
Results **1 - 20** of **10,514**

...

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



how to find patterns

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

...

View (previous 20 | next 20) (20 | 50 | 100 | 250 | 500)

→ Formalization: **enumeration algorithms**

Formalizing Enumeration Algorithms

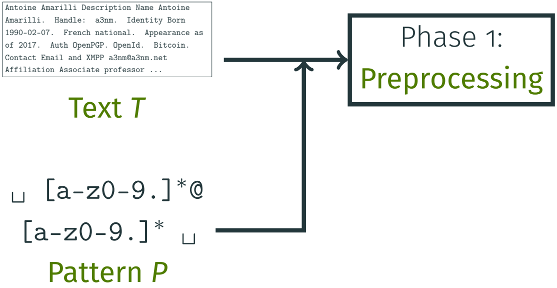
```
Antoine Amarilli Description Name Antoine
Amarilli.  Handle:  a3nm.  Identity Born
1990-02-07.  French national.  Appearance as
of 2017.  Auth OpenPGP. OpenId.  Bitcoin.
Contact Email and XMPP a3nm@a3nm.net
Affiliation Associate professor ...
```

Text *T*

```
␣ [a-z0-9.]*@
[a-z0-9.]* ␣
```

Pattern *P*

Formalizing 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 ...
```

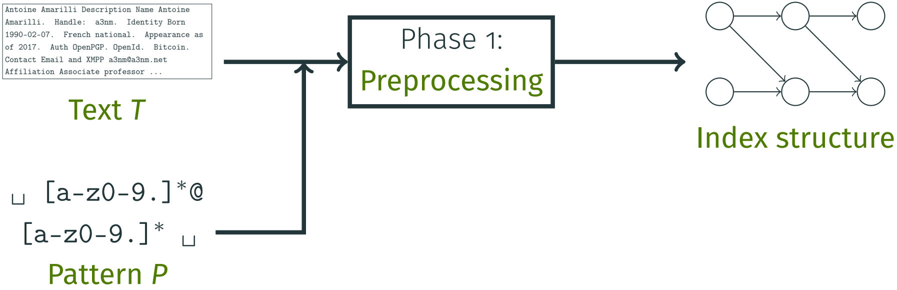
Text *T*

`␣ [a-z0-9.]*@[a-z0-9.]* ␣`

Pattern *P*

Phase 1: Preprocessing

Formalizing 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 ...
```

Text *T*

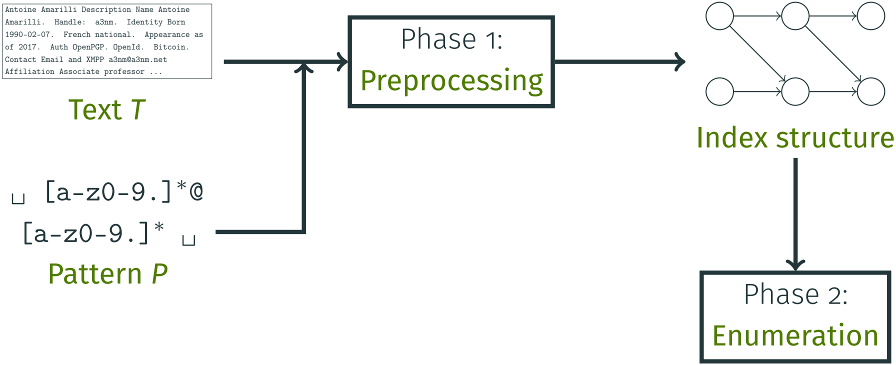
`␣ [a-z0-9.]*@ [a-z0-9.]* ␣`

Pattern *P*

Phase 1: Preprocessing

Index structure

Formalizing Enumeration Algorithms



```
Antoine Amarilli Description Name Antoine
Amarilli.  Handle:  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 ...
```

Text *T*

␣ [a-z0-9.]*@ [a-z0-9.]* ␣

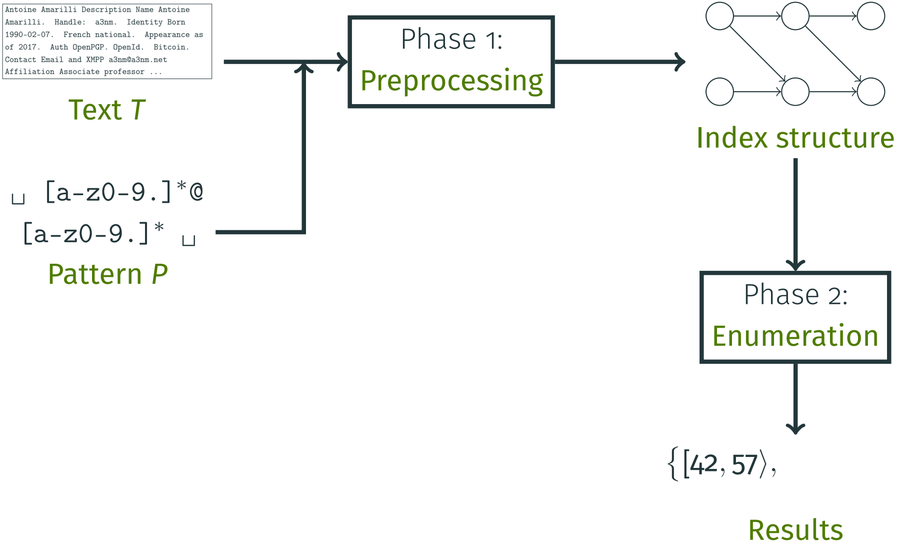
Pattern *P*

Phase 1: Preprocessing

Index structure

Phase 2: Enumeration

Formalizing 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 ...
```

Text T

`␣[a-z0-9.]*@[a-z0-9.]*␣`

Pattern P

Phase 1: Preprocessing

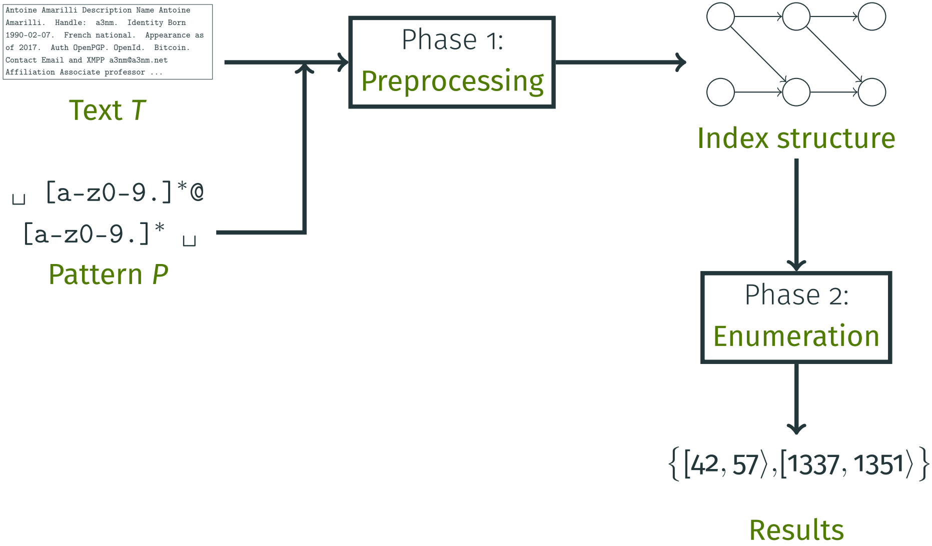
Index structure

Phase 2: Enumeration

$\{[42, 57\rangle,$

Results

Formalizing 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 ...
```

Text *T*

`␣ [a-z0-9.]*@ [a-z0-9.]* ␣`

Pattern *P*

Phase 1:

Preprocessing

Index structure

Phase 2:

Enumeration

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

Results

Formalizing Enumeration Algorithms

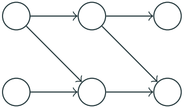
```
Antoine Amarilli Description Name Antoine
Amarilli.  Handle:  a3nm.  Identity Born
1990-02-07.  French national.  Appearance as
of 2017.  Auth OpenPGP. OpenId.  Bitcoin.
Contact Email and XMPP a3nm@a3nm.net
Affiliation Associate professor ...
```

Text T

```
␣ [a-z0-9.]*@
 [a-z0-9.]* ␣
```

Pattern P

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 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 Paris, 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 Paris, 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

$$P := \texttt{␣ [a-z0-9.]* @ [a-z0-9.]* ␣}$$

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 Paris, 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**?

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 Paris, 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**

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 Paris, 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

$$P := \textvisiblespace\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \textvisiblespace$$

- 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 Paris, 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

$$P := \sqcup\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \sqcup$$

- 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: They only measure the complexity as a function of 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: They only measure the complexity as a function of T!

- 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: They only measure the complexity as a function of T!

- Our contribution is:

Theorem

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

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

Automaton Formalism

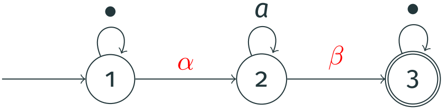
- We use automata that read letters and capture variables

Automaton Formalism

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

Automaton Formalism

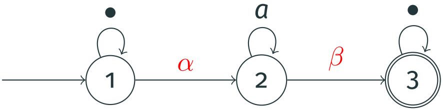
- We use automata that read letters and capture variables
  - → Example: $P := \bullet^* \ \alpha \ a^* \ \beta \ \bullet^*$



Automaton Formalism

- We use automata that read letters and capture variables

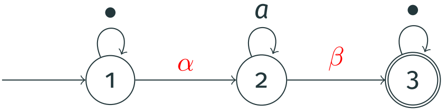
  → Example: $P := \bullet^* \;\; \alpha \;\; a^* \;\; \beta \;\; \bullet^*$



- 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: $P := \bullet^* \;\; \alpha \;\; a^* \;\; \beta \;\; \bullet^*$



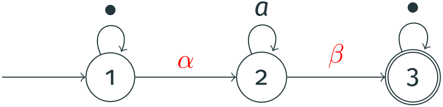
- Semantics of the automaton A:
  - Reads letters from the text
  - Guesses variables at positions in the text

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

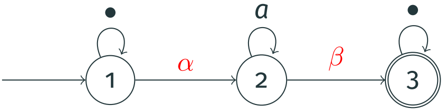
  → Output: tuples $\langle \alpha : i, \beta : j \rangle$ such that
  A has an accepting run reading α at position i and β at j

- 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 α at position i and β 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

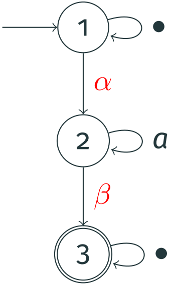
Compute a **product DAG** of the text T and of the automaton A

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

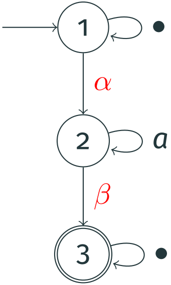


Proof Idea: Product DAG

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

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

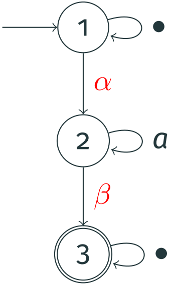
a	a	a	b	a

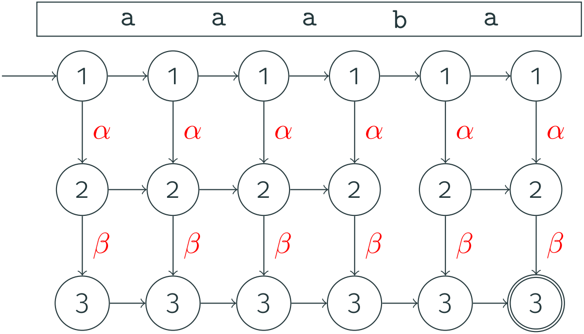


Proof Idea: Product DAG

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

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

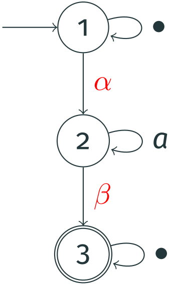


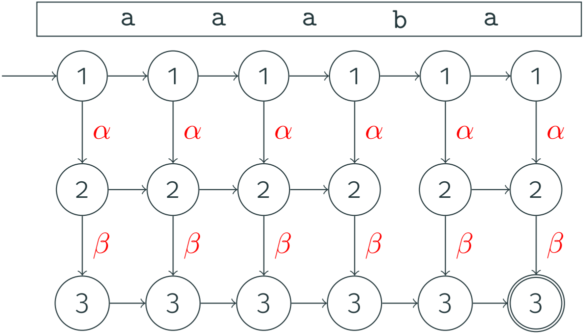


Proof Idea: Product DAG

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

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



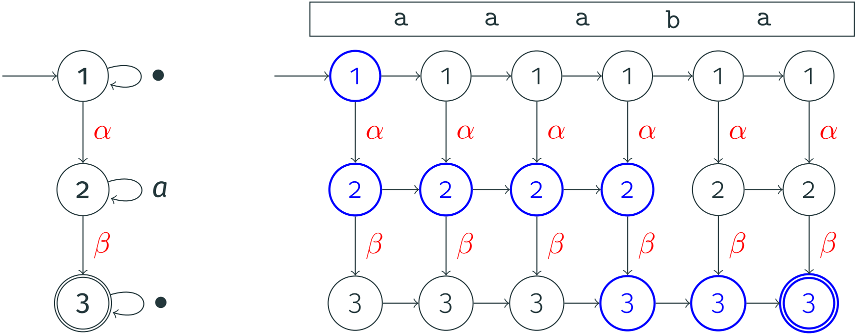


→ Each **path** in the **product DAG** corresponds to a **match**

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 $\langle \alpha : 0, \beta : 3 \rangle$

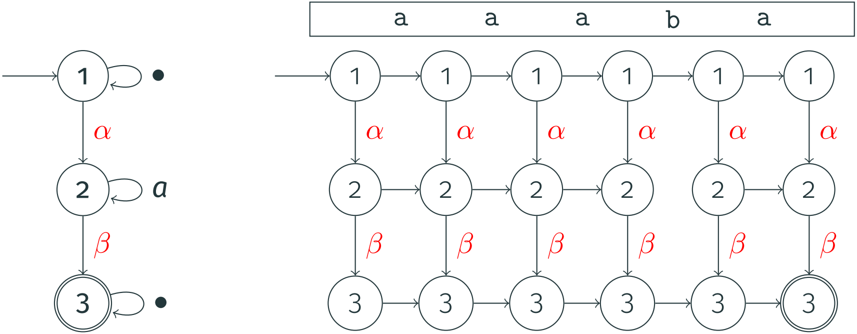


→ Each path in the product DAG corresponds to a match

Proof Idea: Product DAG

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

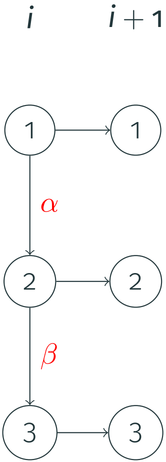
Example: Text $T :=$ `aaaba` and $P := \bullet^* \ \alpha \ a^* \ \beta \ \bullet^*$,



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

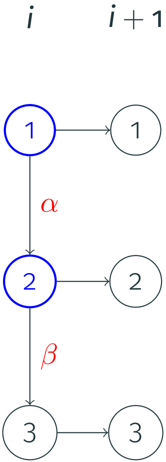
→ **Challenge:** Enumerate paths but avoid **duplicate matches** and do not **waste time** to ensure constant delay

Proof idea: on-the-fly computation to avoid duplicates



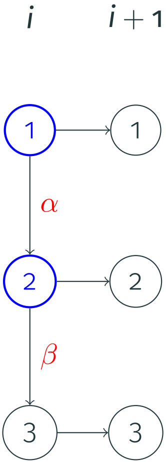
- We are at a position i and set of states in blue

Proof idea: on-the-fly computation to avoid duplicates



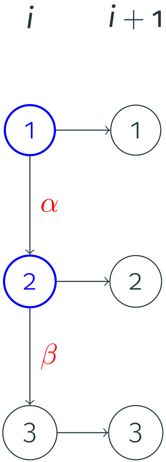
- We are at a position i and set of states in blue

Proof idea: on-the-fly computation to avoid duplicates



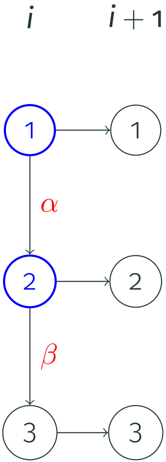
- We are at a position i and set of states in blue
- Partition tuples based on the set S of variables assigned at the current position

Proof idea: on-the-fly computation to avoid duplicates



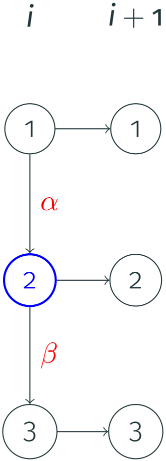
- We are at a position i and set of states in blue
- Partition tuples based on the set S of variables assigned at the current position
- For each S, consider the set of states where we can be at $i+1$ when reading S at i

Proof idea: on-the-fly computation to avoid duplicates



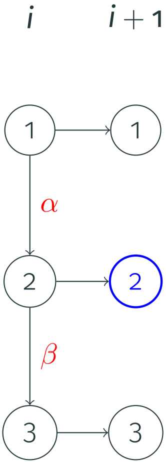
- We are at a position i and set of states in blue
- Partition tuples based on the set S of variables assigned at the current position
- For each S, consider the set of states where we can be at $i+1$ when reading S at i
    - Example: $S = \{\alpha\}$

Proof idea: on-the-fly computation to avoid duplicates



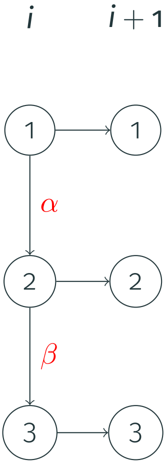
- We are at a position i and set of states in blue
- Partition tuples based on the set S of variables assigned at the current position
- For each S, consider the set of states where we can be at $i+1$ when reading S at i
  - Example: $S = \{\alpha\}$

Proof idea: on-the-fly computation to avoid duplicates



- We are at a position i and set of states in blue
- Partition tuples based on the set S of variables assigned at the current position
- For each S, consider the set of states where we can be at $i+1$ when reading S at i
    - Example: $S = \{\alpha\}$

Proof idea: on-the-fly computation to avoid duplicates



- We are at a **position** i and **set of states** in blue
- Partition tuples based on the **set S of variables** assigned at the current position
- For each S, consider the **set of states** where we can be at $i+1$ when reading S at i
  - Example: $S = \{\alpha\}$

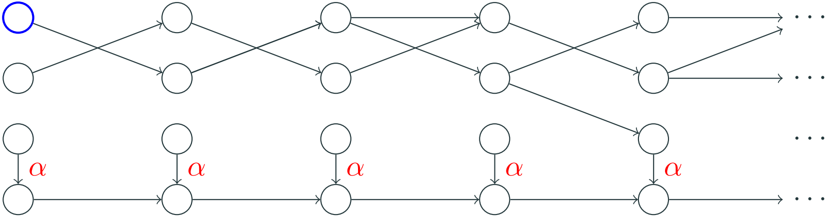
→ We must have **preprocessed** the DAG to make sure that we can always finish the run

Proof idea: jump pointers to save time

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

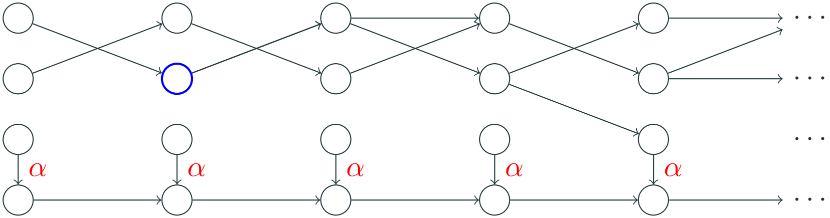
Proof idea: jump pointers to save time

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



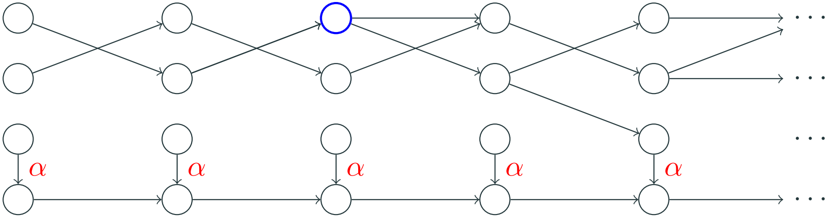
Proof idea: jump pointers to save time

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



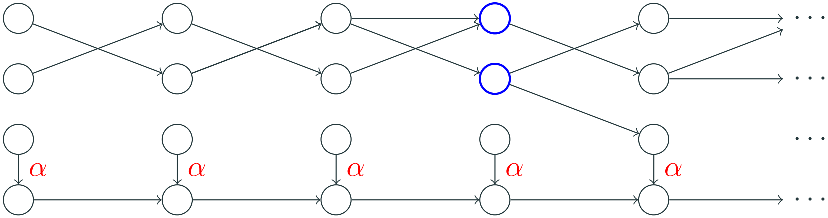
Proof idea: jump pointers to save time

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



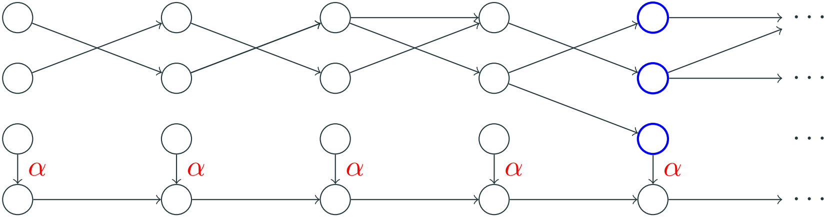
Proof idea: jump pointers to save time

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



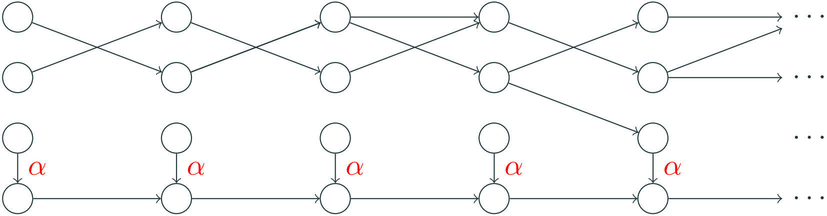
Proof idea: jump pointers to save time

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



Proof idea: jump pointers to save time

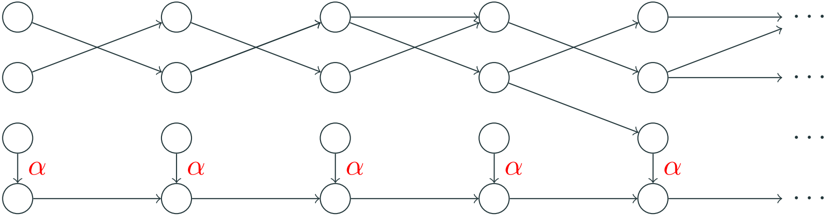
- Issue: When we can't assign variables, we do not make progress



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

Proof idea: jump pointers to save time

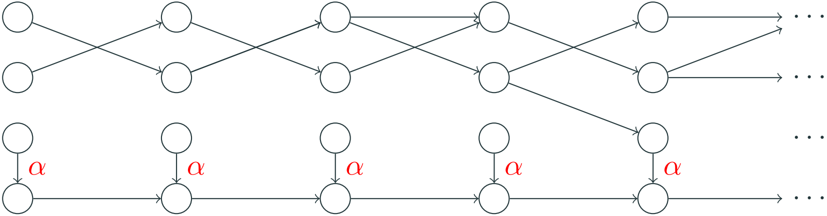
- Issue: When we can't assign variables, we do not make progress



- 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 idea: jump pointers to save time

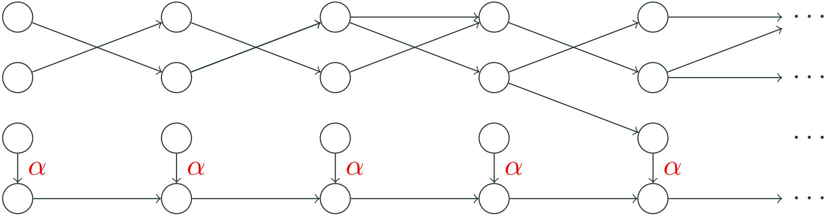
- Issue: When we can't assign variables, we do not make progress



- 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 idea: jump pointers to save time

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

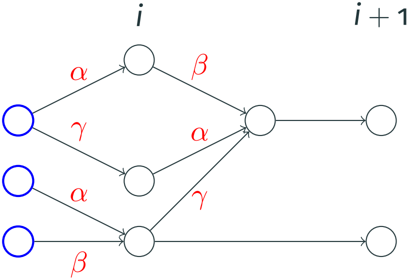


- Idea: Directly jump to the reachable states at the next position where we can assign a variable

- 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 $\leq |A|$)

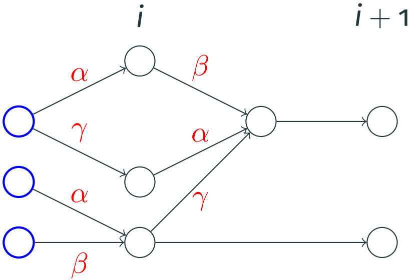
Proof idea: flashlight search

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



Proof idea: flashlight search

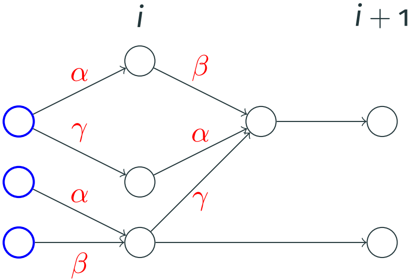
- Issue: Finding which variable sets we can assign at position i?

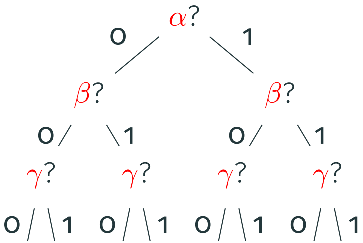


- Idea: Explore a decision tree on the variables (built on the fly)

Proof idea: flashlight search

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

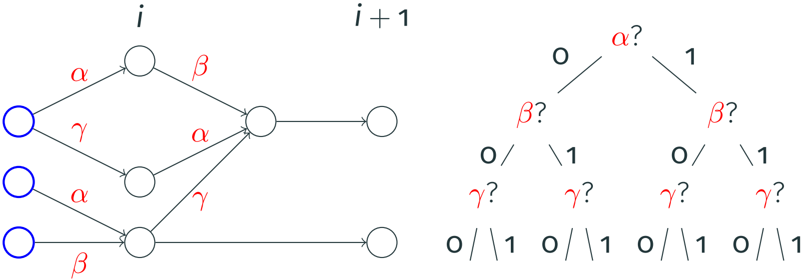




- Idea: Explore a decision tree on the variables (built on the fly)

Proof idea: flashlight search

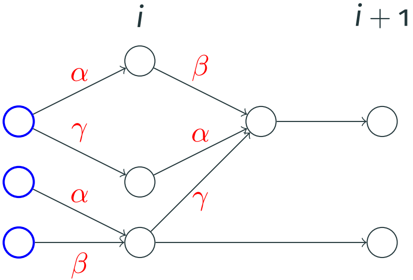
- **Issue:** Finding which **variable sets** we can assign at position i?

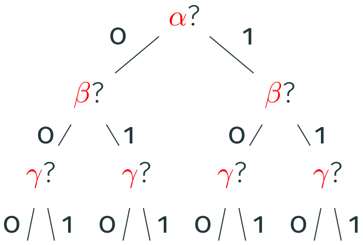


- **Idea:** Explore a **decision tree** on the variables (built on the fly)
- At each decision tree **node**, find the reachable **states** which have **all required variables** (1) and **no forbidden variables** (0)

Proof idea: flashlight search

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





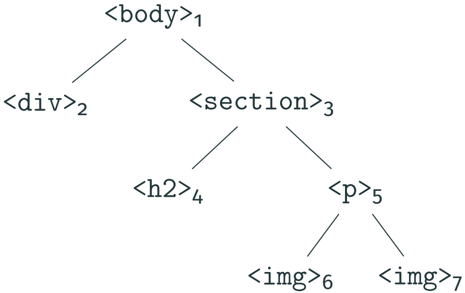
- **Idea:** Explore a **decision tree** on the variables (built on the fly)

- 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

Extension: From Text to Trees

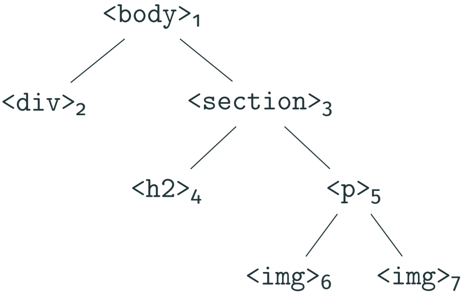
Pattern Matching on Trees

- The data T is no longer text but is now a tree:



Pattern Matching on Trees

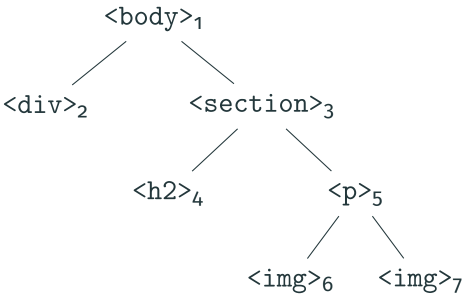
- The **data** *T* is no longer **text** but is now a **tree**:



- The **pattern** *P* asks about the **structure** of the tree:
  *Is there an **h2** header and an **image** in the same section?*

Pattern Matching on Trees

- The data T is no longer text but is now a tree:

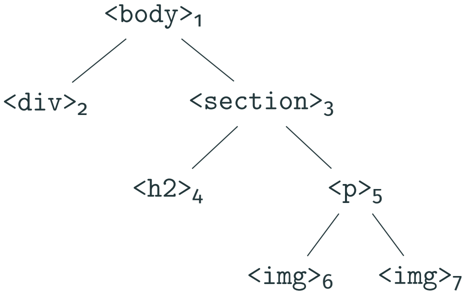


- The pattern P asks about the structure of the tree:
  Is there an h2 header and an image in the same section?

- Results:

Pattern Matching on Trees

- The **data** *T* is no longer **text** but is now a **tree**:

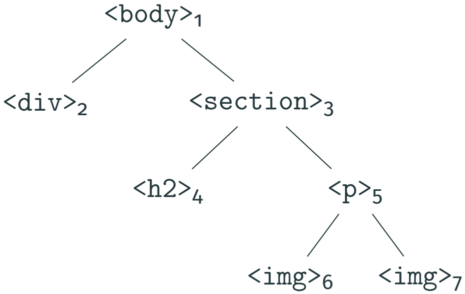


- The **pattern** *P* asks about the **structure** of the tree:
  Is there α: an ***h2*** *header and β: an* ***image*** *in the same section?*

- **Results:**

Pattern Matching on Trees

- The **data** *T* is no longer **text** but is now a **tree**:



- The **pattern** *P* asks about the **structure** of the tree:
  *Is there α: an **h2** header and β: an **image** in the same section?*

- **Results:** $\langle \alpha : 4, \beta : 6 \rangle, \langle \alpha : 4, \beta : 7 \rangle$

Definitions and Results on Trees

- Tree patterns P can be written as a kind of tree automaton…

Definitions and Results on Trees

- Tree patterns P can be written as a kind of tree automaton…
- Existing work has studied this problem and shown:

Definitions and Results on Trees

- Tree patterns P can be written as a kind of tree automaton...
- Existing work has studied this problem and shown:

Theorem [Bagan, 2006]

We can find all matches on a tree T of a tree pattern P (with constantly many capture variables) with:

- *Preprocessing linear in T*
- *Delay constant in T*

Definitions and Results on Trees

- Tree patterns *P* can be written as a kind of tree automaton...
- Existing work has studied this problem and shown:

Theorem [Bagan, 2006]

We can find all matches on a tree ***T*** *of a tree pattern* ***P***
(with constantly many capture variables) with:

- *Preprocessing linear in* ***T*** *and exponential in P*
- *Delay constant in* ***T*** *and exponential in P*

- Again, this only measures the complexity in *T*! We show:

Definitions and Results on Trees

- Tree patterns P can be written as a kind of tree automaton...
- Existing work has studied this problem and shown:

Theorem [Bagan, 2006]

We can find all matches on a tree T *of a tree pattern* P *(with constantly many capture variables) with:*

- *Preprocessing linear in* T *and exponential in* P
- *Delay constant in* T *and exponential in* P

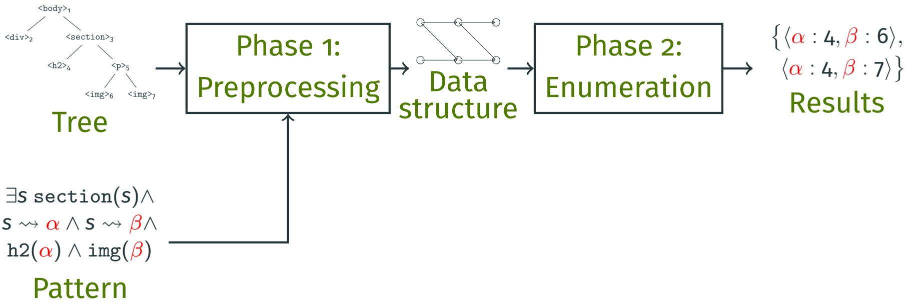
- Again, this only measures the complexity in T! We show:

Theorem [Amarilli et al., 2019]

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

Proof Idea for Trees: Structure

Similar structure to the previous proof, but with a **circuit**:



Tree

Phase 1: Preprocessing

Data structure

Phase 2: Enumeration

$\{\langle \alpha : 4, \beta : 6 \rangle,$
$\langle \alpha : 4, \beta : 7 \rangle\}$

Results

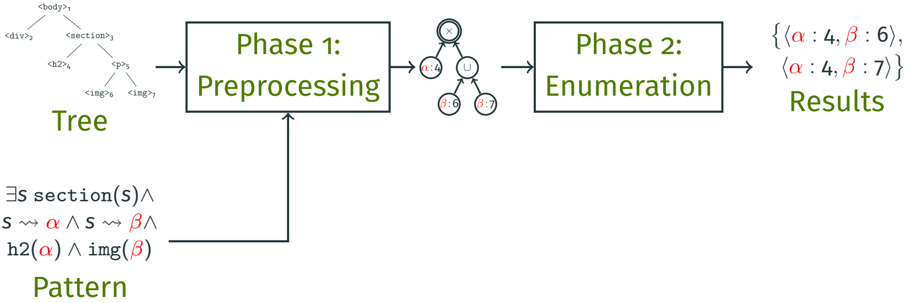
$\exists s\ \texttt{section}(s) \wedge$
$s \rightsquigarrow \alpha \wedge s \rightsquigarrow \beta \wedge$
$\texttt{h2}(\alpha) \wedge \texttt{img}(\beta)$

Pattern

Proof Idea for Trees: Structure

Similar structure to the previous proof, but with a circuit:

- Preprocessing: Compute a circuit representation of the answers
- Enumeration: Apply a generic algorithm on the circuit



Tree

Phase 1: Preprocessing

Phase 2: Enumeration

$\{\langle \alpha : 4, \beta : 6\rangle, \langle \alpha : 4, \beta : 7\rangle\}$

Results

$\exists s\ \texttt{section}(s) \wedge s \rightsquigarrow \alpha \wedge s \rightsquigarrow \beta \wedge \texttt{h2}(\alpha) \wedge \texttt{img}(\beta)$

Pattern

Proof Idea for Trees: Set Circuits

A set circuit represents a set of answers to a pattern $P(\alpha, \beta)$

Proof Idea for Trees: Set Circuits

A set circuit represents a set of answers to a pattern $P(\alpha, \beta)$

- Singleton α:6 → *"the variable α is mapped to node* 6*"*

Proof Idea for Trees: Set Circuits

A **set circuit** represents a **set of answers** to a pattern $P(\alpha, \beta)$

- **Singleton** $\alpha : 6 \rightarrow$ *"the variable α is mapped to node* 6*"*
- **Tuple** $\langle \alpha : 4, \beta : 6 \rangle$: tuple of singletons

Proof Idea for Trees: Set Circuits

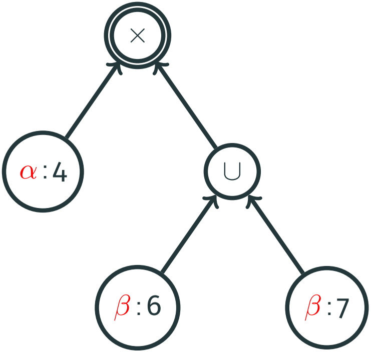
A set circuit represents a set of answers to a pattern $P(\alpha, \beta)$

- Singleton $\alpha : 6$ → *"the variable α is mapped to node* 6*"*
- Tuple $\langle \alpha : 4, \beta : 6 \rangle$: tuple of singletons
- The circuit captures a set of tuples, e.g., $\{\langle \alpha : 4, \beta : 6 \rangle, \langle \alpha : 4, \beta : 7 \rangle\}$

Proof Idea for Trees: Set Circuits

A **set circuit** represents a **set of answers** to a pattern $P(\alpha, \beta)$

- **Singleton** $\alpha : 6$ → *“the variable α is mapped to node* 6*”*
- **Tuple** $\langle \alpha : 4, \beta : 6 \rangle$: tuple of singletons
- The circuit captures a **set** of tuples, e.g., $\{\langle \alpha : 4, \beta : 6 \rangle, \langle \alpha : 4, \beta : 7 \rangle\}$

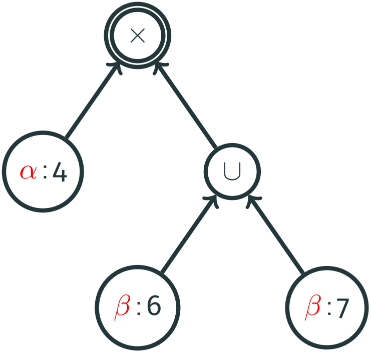


Three kinds of **set-valued gates**:

Proof Idea for Trees: Set Circuits

A **set circuit** represents a **set of answers** to a pattern $P(\alpha, \beta)$

- **Singleton** $\alpha : 6$ → *“the variable* α *is mapped to node* 6*”*
- **Tuple** $\langle \alpha : 4, \beta : 6 \rangle$: tuple of singletons
- The circuit captures a **set** of tuples, e.g., $\{\langle \alpha : 4, \beta : 6 \rangle, \langle \alpha : 4, \beta : 7 \rangle\}$



Three kinds of **set-valued gates**:

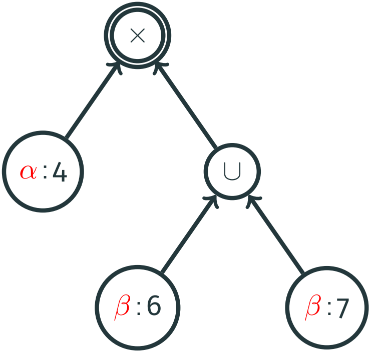
- **Variable gate** $\alpha : 4$:

  → captures $\{\langle \alpha : 4 \rangle\}$

Proof Idea for Trees: Set Circuits

A **set circuit** represents a **set of answers** to a pattern $P(\alpha, \beta)$

- **Singleton** $\alpha : 6 \rightarrow$ *“the variable α is mapped to node* 6*”*
- **Tuple** $\langle \alpha : 4, \beta : 6 \rangle$: tuple of singletons
- The circuit captures a **set** of tuples, e.g., $\{\langle \alpha : 4, \beta : 6 \rangle, \langle \alpha : 4, \beta : 7 \rangle\}$



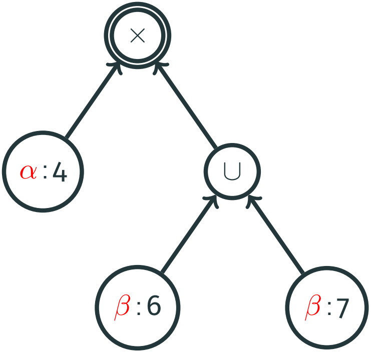
Three kinds of **set-valued gates**:

- **Variable gate** $\alpha : 4$:
  - $\rightarrow$ captures $\{\langle \alpha : 4 \rangle\}$
- **Union gate** ∪ :
  - $\rightarrow$ union of sets of tuples

Proof Idea for Trees: Set Circuits

A **set circuit** represents a **set of answers** to a pattern $P(\alpha, \beta)$

- **Singleton** $\alpha:6$ → *“the variable α is mapped to node* 6*”*
- **Tuple** $\langle \alpha:4, \beta:6 \rangle$: tuple of singletons
- The circuit captures a **set** of tuples, e.g., $\{\langle \alpha:4, \beta:6 \rangle, \langle \alpha:4, \beta:7 \rangle\}$



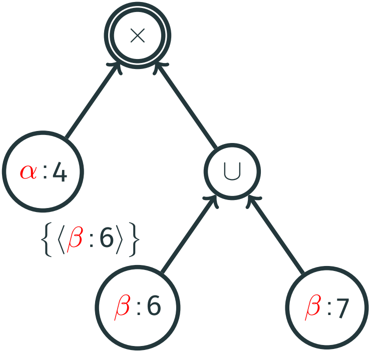
Three kinds of **set-valued gates**:

- **Variable gate** $\alpha:4$:
  → captures $\{\langle \alpha:4 \rangle\}$
- **Union gate** $\cup$:
  → union of sets of tuples
- **Product gate** $\times$:
  → relational product

Proof Idea for Trees: Set Circuits

A **set circuit** represents a **set of answers** to a pattern $P(\alpha, \beta)$

- **Singleton** $\alpha : 6$ → *“the variable α is mapped to node* 6*”*
- **Tuple** $\langle \alpha : 4, \beta : 6 \rangle$: tuple of singletons
- The circuit captures a **set** of tuples, e.g., $\{\langle \alpha : 4, \beta : 6 \rangle, \langle \alpha : 4, \beta : 7 \rangle\}$



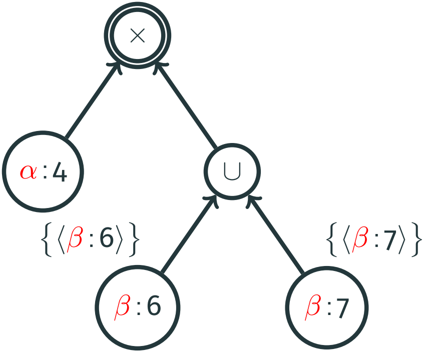
Three kinds of **set-valued gates**:

- **Variable gate** $\alpha : 4$:
  → captures $\{\langle \alpha : 4 \rangle\}$
- **Union gate** $\cup$:
  → union of sets of tuples
- **Product gate** $\times$:
  → relational product

Proof Idea for Trees: Set Circuits

A set circuit represents a set of answers to a pattern $P(\alpha, \beta)$

- Singleton $\alpha : 6$ → *“the variable α is mapped to node* 6*”*
- Tuple $\langle \alpha : 4, \beta : 6 \rangle$: tuple of singletons
- The circuit captures a set of tuples, e.g., $\{\langle \alpha : 4, \beta : 6 \rangle, \langle \alpha : 4, \beta : 7 \rangle\}$


×
$\alpha : 4$
∪
$\{\langle \beta : 6 \rangle\}$
$\{\langle \beta : 7 \rangle\}$
$\beta : 6$
$\beta : 7$

Three kinds of set-valued gates:

- Variable gate $\alpha : 4$:

  → captures $\{\langle \alpha : 4 \rangle\}$
- Union gate ∪ :

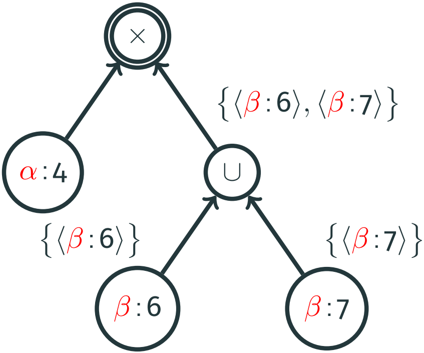
  → union of sets of tuples
- Product gate × :

  → relational product

Proof Idea for Trees: Set Circuits

A **set circuit** represents a **set of answers** to a pattern $P(\alpha, \beta)$

- **Singleton** $\alpha : 6$ → *"the variable α is mapped to node 6"*
- **Tuple** $\langle \alpha : 4, \beta : 6 \rangle$: tuple of singletons
- The circuit captures a **set** of tuples, e.g., $\{\langle \alpha : 4, \beta : 6 \rangle, \langle \alpha : 4, \beta : 7 \rangle\}$



$\times$
$\{\langle \beta : 6 \rangle, \langle \beta : 7 \rangle\}$
$\alpha : 4$
$\cup$
$\{\langle \beta : 6 \rangle\}$
$\{\langle \beta : 7 \rangle\}$
$\beta : 6$
$\beta : 7$

Three kinds of **set-valued gates**:

- **Variable gate** $\alpha : 4$:

  → captures $\{\langle \alpha : 4 \rangle\}$

- **Union gate** $\cup$:

  → union of sets of tuples

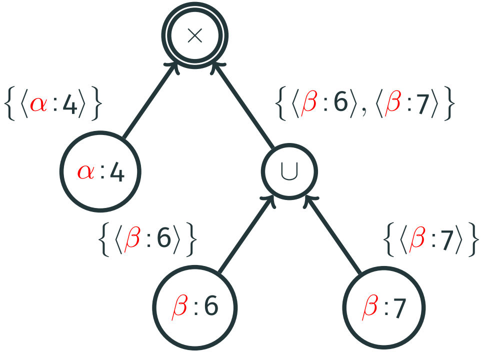
- **Product gate** $\times$:

  → relational product

Proof Idea for Trees: Set Circuits

A **set circuit** represents a **set of answers** to a pattern $P(\alpha, \beta)$

- **Singleton** $\alpha : 6$ → *“the variable α is mapped to node 6”*
- **Tuple** $\langle \alpha : 4, \beta : 6 \rangle$: tuple of singletons
- The circuit captures a **set** of tuples, e.g., $\{\langle \alpha : 4, \beta : 6 \rangle, \langle \alpha : 4, \beta : 7 \rangle\}$



$\times$
$\{\langle \alpha : 4 \rangle\}$
$\{\langle \beta : 6 \rangle, \langle \beta : 7 \rangle\}$
$\alpha : 4$
$\cup$
$\{\langle \beta : 6 \rangle\}$
$\{\langle \beta : 7 \rangle\}$
$\beta : 6$
$\beta : 7$

Three kinds of **set-valued gates**:

- **Variable gate** $\alpha : 4$:

  → captures $\{\langle \alpha : 4 \rangle\}$
- **Union gate** $\cup$:

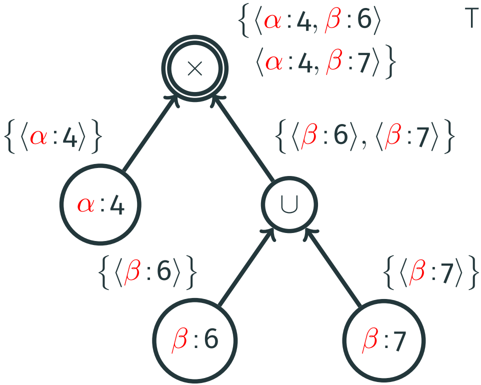
  → union of sets of tuples
- **Product gate** $\times$:

  → relational product

Proof Idea for Trees: Set Circuits

A **set circuit** represents a **set of answers** to a pattern $P(\alpha, \beta)$

- **Singleton** $\alpha : 6 \rightarrow$ *“the variable α is mapped to node 6”*
- **Tuple** $\langle \alpha : 4, \beta : 6 \rangle$: tuple of singletons
- The circuit captures a **set** of tuples, e.g., $\{\langle \alpha : 4, \beta : 6 \rangle, \langle \alpha : 4, \beta : 7 \rangle\}$



$\{\langle \alpha : 4, \beta : 6 \rangle$
$\langle \alpha : 4, \beta : 7 \rangle\}$
$\times$
$\{\langle \alpha : 4 \rangle\}$
$\{\langle \beta : 6 \rangle, \langle \beta : 7 \rangle\}$
$\alpha : 4$
$\cup$
$\{\langle \beta : 6 \rangle\}$
$\{\langle \beta : 7 \rangle\}$
$\beta : 6$
$\beta : 7$

Three kinds of **set-valued gates**:

- **Variable gate** $\alpha : 4$:

  $\rightarrow$ captures $\{\langle \alpha : 4 \rangle\}$

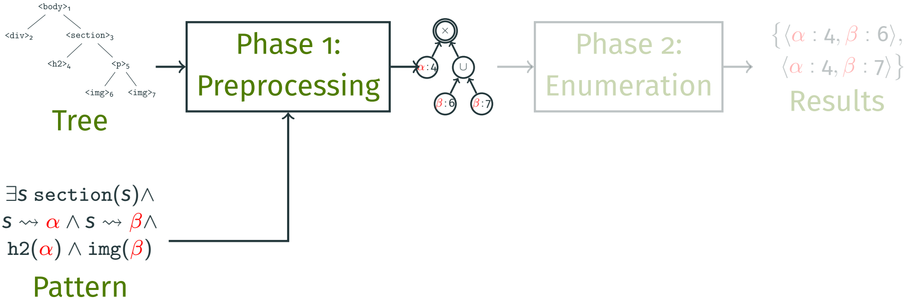
- **Union gate** $\cup$:

  $\rightarrow$ union of sets of tuples

- **Product gate** $\times$:

  $\rightarrow$ relational product

Proof Idea for Trees: Results



Tree

Phase 1: Preprocessing

Phase 2: Enumeration

$\{\langle \alpha : 4, \beta : 6\rangle, \langle \alpha : 4, \beta : 7\rangle\}$

Results

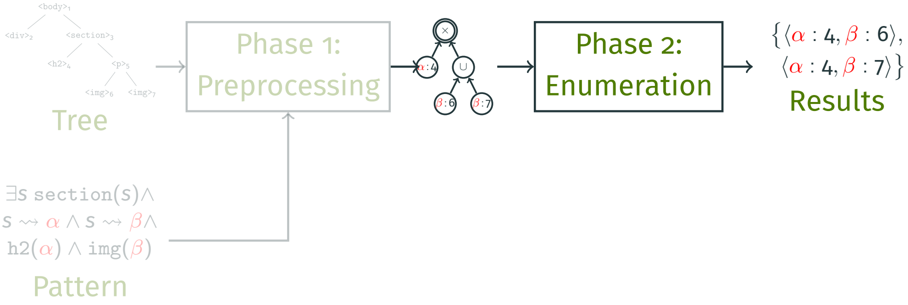
$\exists s\ \mathtt{section}(s) \land s \rightsquigarrow \alpha \land s \rightsquigarrow \beta \land \mathtt{h2}(\alpha) \land \mathtt{img}(\beta)$

Pattern

Theorem

*For any **tree automaton** A with capture variables $\alpha_1, \ldots, \alpha_k$, given a **tree** T, we can build in $O(|T| \times |A|)$ a **set circuit** capturing exactly the set of tuples $\{\langle \alpha_1 : n_1, \ldots, \alpha_k : n_k\rangle\}$ in the output of A on T*

Proof Idea for Trees: Results


<body>$_1$
<div>$_2$
<section>$_3$
<h2>$_4$
<p>$_5$
<img>$_6$
<img>$_7$
Tree
$\exists s$ section$(s) \wedge$
$s \rightsquigarrow \alpha \wedge s \rightsquigarrow \beta \wedge$
h2$(\alpha) \wedge$ img(β)
Pattern
Phase 1: Preprocessing
$\times$
α:4
$\cup$
β:6
β:7
Phase 2: Enumeration
$\{\langle\alpha : 4, \beta : 6\rangle,$
$\langle\alpha : 4, \beta : 7\rangle\}$
Results

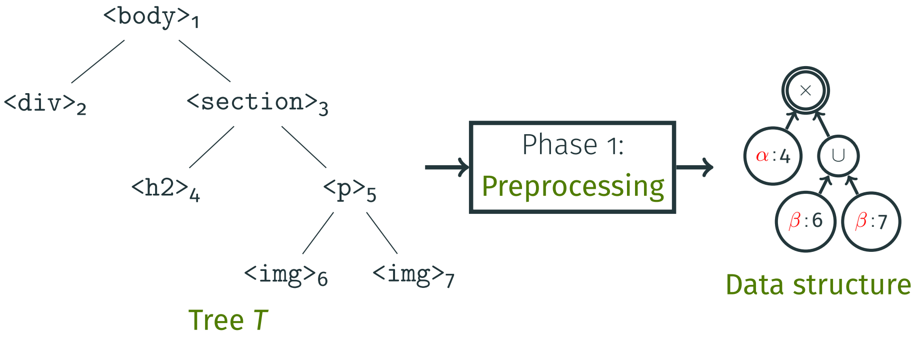
Theorem

*Given a set circuit **satisfying some conditions**, we can enumerate all tuples that it captures with linear preprocessing and constant delay*

E.g., for $\{\langle\alpha : 4, \beta : 6\rangle, \langle\alpha : 4, \beta : 7\rangle\}$: enumerate $\langle\alpha : 4, \beta : 6\rangle$ then $\langle\alpha : 4, \beta : 7\rangle$

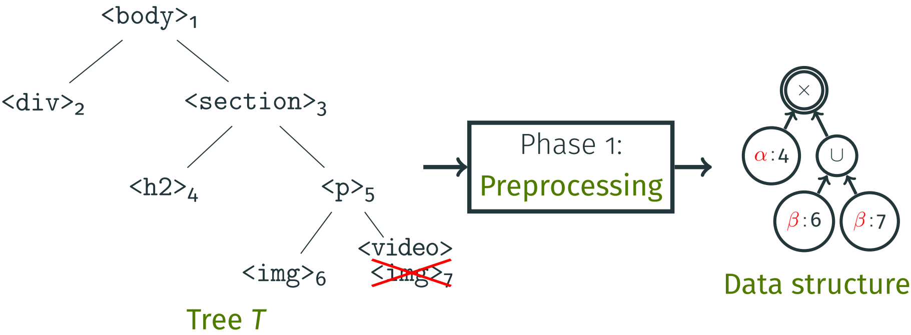
Extension: Supporting Updates

Updates



- The input data can be **modified** after the preprocessing

Updates



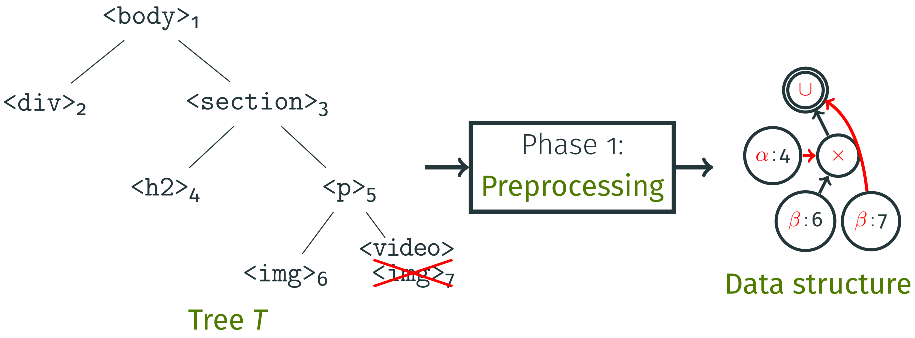
Tree T

Phase 1:
Preprocessing

Data structure

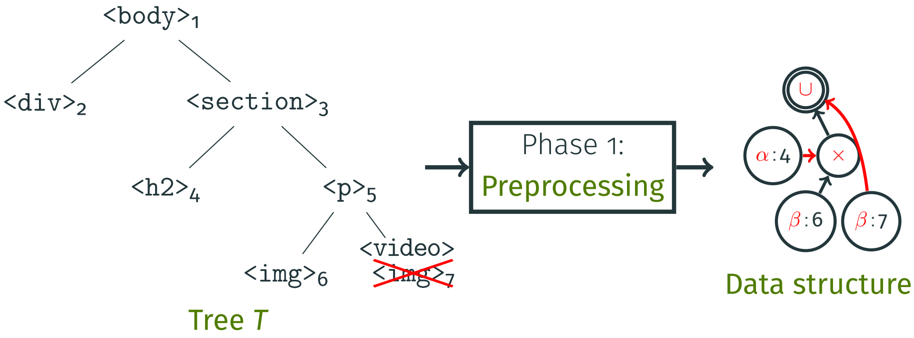
- The input data can be **modified** after the preprocessing

Updates



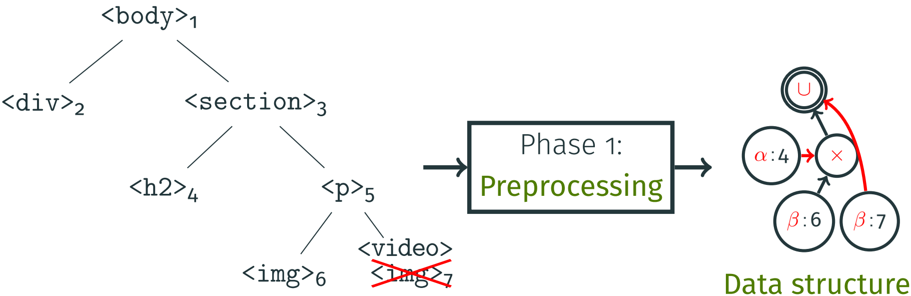
- The input data can be **modified** after the preprocessing

Updates



- The input data can be **modified** after the preprocessing
- If this happen, we must rerun the **preprocessing** from scratch

Updates



Tree T

Data structure

- The input data can be **modified** after the preprocessing
- If this happen, we must rerun the **preprocessing** from scratch

→ Can we **do better**?

Results on dynamic trees

All these results are on data complexity in T (for a fixed pattern):

Work	Data	Preproc.	Delay	Updates
[Bagan, 2006], [Kazana and Segoufin, 2013]	trees	$O(T)$	$O(1)$	$O(T)$

Results on dynamic trees

All these results are on data complexity in T (for a fixed pattern):

Work	Data	Preproc.	Delay	Updates
[Bagan, 2006], [Kazana and Segoufin, 2013]	trees	$O(T)$	$O(1)$	$O(T)$
[Losemann and Martens, 2014]	trees	$O(T)$	$O(\log^2 T)$	$O(\log^2 T)$

Results on dynamic trees

All these results are on data complexity in T (for a fixed pattern):

Work	Data	Preproc.	Delay	Updates
[Bagan, 2006], [Kazana and Segoufin, 2013]	trees	$O(T)$	$O(1)$	$O(T)$
[Losemann and Martens, 2014]	trees	$O(T)$	$O(\log^2 T)$	$O(\log^2 T)$
[Losemann and Martens, 2014]	text	$O(T)$	$O(\log T)$	$O(\log T)$

Results on dynamic trees

All these results are on data complexity in T (for a fixed pattern):

Work	Data	Preproc.	Delay	Updates
[Bagan, 2006], [Kazana and Segoufin, 2013]	trees	$O(T)$	$O(1)$	$O(T)$
[Losemann and Martens, 2014]	trees	$O(T)$	$O(\log^2 T)$	$O(\log^2 T)$
[Losemann and Martens, 2014]	text	$O(T)$	$O(\log T)$	$O(\log T)$
[Niewerth and Segoufin, 2018]	text	$O(T)$	$O(1)$	$O(\log T)$

Results on dynamic trees

All these results are on data complexity in T (for a fixed pattern):

Work	Data	Preproc.	Delay	Updates
[Bagan, 2006], [Kazana and Segoufin, 2013]	trees	$O(T)$	$O(1)$	$O(T)$
[Losemann and Martens, 2014]	trees	$O(T)$	$O(\log^2 T)$	$O(\log^2 T)$
[Losemann and Martens, 2014]	text	$O(T)$	$O(\log T)$	$O(\log T)$
[Niewerth and Segoufin, 2018]	text	$O(T)$	$O(1)$	$O(\log T)$
[Amarilli et al., 2019]	trees	$O(T)$	$O(1)$	$O(\log T)$

Extension: Connection to Circuits

Connections with Boolean circuits

- Mapping DAGs and set circuits can be seen as variants of Boolean circuits

Connections with Boolean circuits

- Mapping DAGs and set circuits can be seen as variants of Boolean circuits
- The answers to enumerate are their satisfying assignments

Connections with Boolean circuits

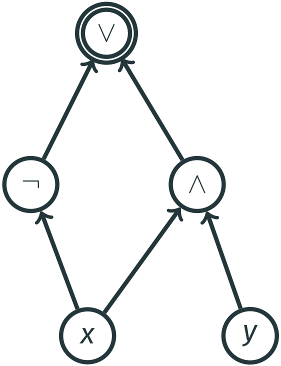
- Mapping DAGs and set circuits can be seen as variants of Boolean circuits
- The answers to enumerate are their satisfying assignments
- These circuits fall in restricted circuit classes that allow for efficient enumeration

Connections with Boolean circuits

- Mapping DAGs and set circuits can be seen as variants of Boolean circuits
- The answers to enumerate are their satisfying assignments
- These circuits fall in restricted circuit classes that allow for efficient enumeration

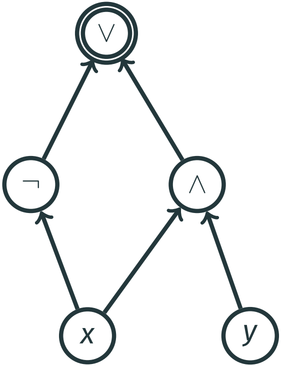
→ Task: Given a Boolean circuit, how to efficiently enumerate its satisfying valuations?

Boolean circuits



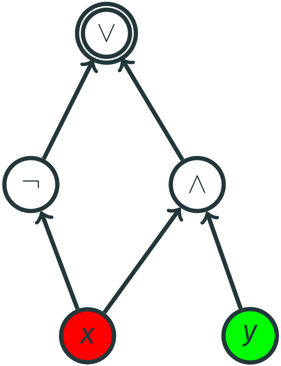
- Directed acyclic graph of **gates**
- **Output** gate: ◎
- **Variable** gates: x
- **Internal** gates: $\lor$ $\land$ $\neg$

Boolean circuits



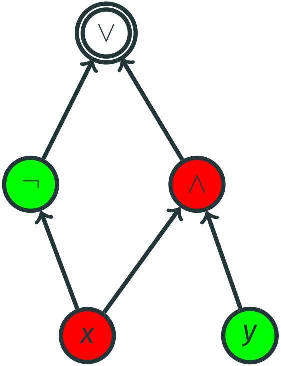
- Directed acyclic graph of **gates**
- **Output** gate: ◎
- **Variable** gates: ⓧ
- **Internal** gates: $\vee$ $\wedge$ $\neg$
- **Valuation**: function from variables to $\{0, 1\}$
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$...

Boolean circuits



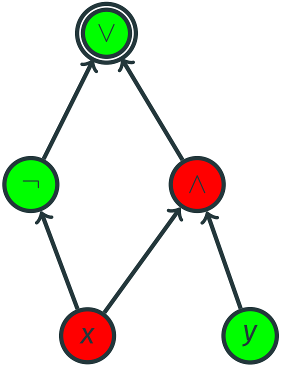
- Directed acyclic graph of **gates**
- **Output** gate: ◎
- **Variable** gates: x
- **Internal** gates: $\lor$ $\land$ $\neg$
- **Valuation**: function from variables to $\{0, 1\}$
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$...

Boolean circuits



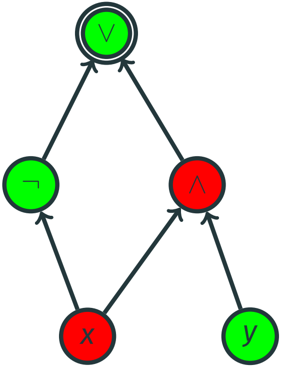
- Directed acyclic graph of **gates**
- **Output** gate: ◎
- **Variable** gates: x
- **Internal** gates: $\lor$ $\land$ $\neg$
- **Valuation**: function from variables to $\{0, 1\}$
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$...

Boolean circuits



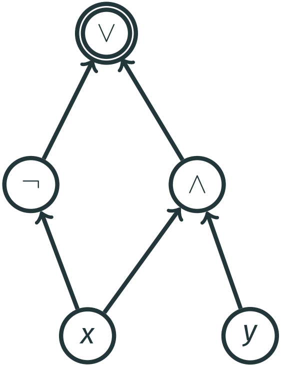
- Directed acyclic graph of **gates**
- **Output** gate: ◎
- **Variable** gates: x
- **Internal** gates: $\lor$ $\land$ $\neg$
- **Valuation**: function from variables to $\{0, 1\}$
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$... mapped to 1

Boolean circuits



- Directed acyclic graph of **gates**
- **Output** gate: ◎
- **Variable** gates: x
- **Internal** gates: $\vee$ $\wedge$ $\neg$
- **Valuation**: function from variables to $\{0, 1\}$
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$... mapped to 1
- **Assignment**: set of variables mapped to 1
  Example: $S_\nu = \{y\}$; more concise than ν

Boolean circuits



- Directed acyclic graph of **gates**
- **Output** gate: ◎
- **Variable** gates: x
- **Internal** gates: ∨ ∧ ¬
- **Valuation**: function from variables to $\{0, 1\}$
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$... mapped to **1**
- **Assignment**: set of variables mapped to 1
  Example: $S_\nu = \{y\}$; more concise than ν

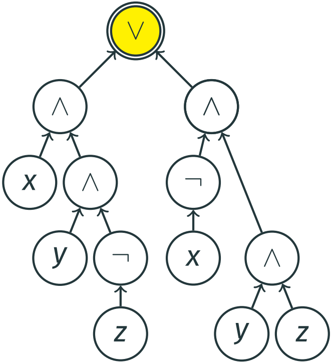
Our task: Enumerate all **satisfying assignments** of an input circuit

Circuit restrictions

d-DNNF:

- $\vee$ are all **deterministic**:

The inputs are **mutually exclusive** (= no valuation ν makes two inputs simultaneously evaluate to 1)



Circuit restrictions

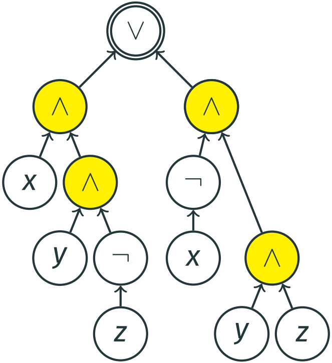
d-DNNF:

- $\vee$ are all **deterministic**:

The inputs are **mutually exclusive** (= no valuation ν makes two inputs simultaneously evaluate to 1)

- $\wedge$ are all **decomposable**:

The inputs are **independent** (= no variable x has a path to two different inputs)



Circuit restrictions

d-DNNF:

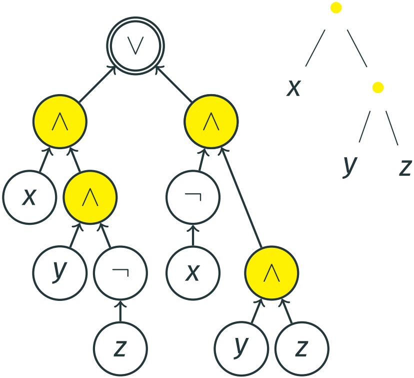
- ∨ are all **deterministic**:

The inputs are **mutually exclusive** (= no valuation ν makes two inputs simultaneously evaluate to 1)

- ∧ are all **decomposable**:

The inputs are **independent** (= no variable x has a path to two different inputs)

v-tree: ∧-gates follow a **tree** on the variables



Main results

Theorem

*Given a **d-DNNF circuit** C with a **v-tree** T, we can enumerate its **satisfying assignments** with preprocessing **linear in** $|C| + |T|$ and delay **linear in each assignment***

Main results

Theorem

Given a d-DNNF circuit C with a v-tree T, we can enumerate its satisfying assignments with preprocessing linear in $|C| + |T|$ and delay linear in each assignment

Also: restrict to assignments of **constant size** $k \in \mathbb{N}$ (at most k variables are set to 1):

Theorem

Given a d-DNNF circuit C with a v-tree T, we can enumerate its satisfying assignments of size $\leq k$ with preprocessing linear in $|C| + |T|$ and constant delay

Main results

Theorem

Given a d-DNNF circuit C with a v-tree T, we can enumerate its satisfying assignments with preprocessing linear in $|C| + |T|$ and delay linear in each assignment

Also: restrict to assignments of constant size $k \in \mathbb{N}$ (at most k variables are set to 1):

Theorem

Given a d-DNNF circuit C with a v-tree T, we can enumerate its satisfying assignments of size $\leq k$ with preprocessing linear in $|C| + |T|$ and constant delay

Subtleties: Must complete to a set circuit; memory usage problems

Summary and Future Work

Summary and Future Work

Summary:

- **Problem:** given a text T and a pattern P, enumerate efficiently all matches of P on T

Summary and Future Work

Summary:

- Problem: given a text T and a pattern P, enumerate efficiently all matches of P on T
- Result: we can do this with reasonable complexity in P and with linear preprocessing and constant delay in T

Summary and Future Work

Summary:

- Problem: given a text T and a pattern P, enumerate efficiently all matches of P on T
- Result: we can do this with reasonable complexity in P and with linear preprocessing and constant delay in T

Extensions:

- Enumeration on trees rather than words
- Handling updates to the underlying data
- Enumerating satisfying valuations of a circuit

Ongoing research and future work

With P. Bourhis, R. Dupré, M. Niewerth, S. Mengel:

Efficient implementation of the approach

`https://github.com/PoDMR/enum-spanner-rs`

Ongoing research and future work

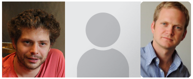
With P. Bourhis, R. Dupré, M. Niewerth, S. Mengel:

Efficient implementation of the approach

`https://github.com/PoDMR/enum-spanner-rs`

With L. Jachiet, M. Muñoz, C. Riveros:

Can we enumerate for **context-free languages**?



Ongoing research and future work

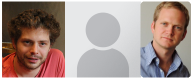
With ***P. Bourhis, R. Dupré, M. Niewerth, S. Mengel****:*

Efficient implementation of the approach

`https://github.com/PoDMR/enum-spanner-rs`

With ***L. Jachiet, M. Muñoz, C. Riveros****:*

Can we enumerate for **context-free languages**?



With ***P. Bourhis, C. Riveros, S. Mengel****:*

Links between **automata** and **circuit classes**?

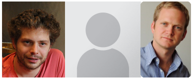
Ongoing research and future work

With P. Bourhis, R. Dupré, M. Niewerth, S. Mengel:

Efficient implementation of the approach

`https://github.com/PoDMR/enum-spanner-rs`

With L. Jachiet, M. Muñoz, C. Riveros:



Can we enumerate for **context-free languages**?

With P. Bourhis, C. Riveros, S. Mengel:

Links between **automata** and **circuit classes**?

With P. Bourhis, L. Jachiet, S. Mengel:

Can we enumerate with **constant memory usage**?

Ongoing research and future work

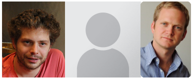
With ***P. Bourhis, R. Dupré, M. Niewerth, S. Mengel:***

Efficient implementation of the approach

`https://github.com/PoDMR/enum-spanner-rs`

With ***L. Jachiet, M. Muñoz, C. Riveros:***

Can we enumerate for **context-free languages**?



With ***P. Bourhis, C. Riveros, S. Mengel:***

Links between **automata** and **circuit classes**?

With ***P. Bourhis, L. Jachiet, S. Mengel:***

Can we enumerate with **constant memory usage**?

With ***B. Kimelfeld, S. Mengel:***

How to enumerate **maximal matches** of a pattern?

Ongoing research and future work

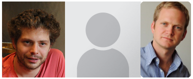
*With **P. Bourhis, R. Dupré, M. Niewerth, S. Mengel**:*

Efficient implementation of the approach

`https://github.com/PoDMR/enum-spanner-rs`

---

*With **L. Jachiet, M. Muñoz, C. Riveros**:*



Can we enumerate for **context-free languages**?

---

*With **P. Bourhis, C. Riveros, S. Mengel**:*

Links between **automata** and **circuit classes**?

---

*With **P. Bourhis, L. Jachiet, S. Mengel**:*

Can we enumerate with **constant memory usage**?

---

*With **B. Kimelfeld, S. Mengel**:*

How to enumerate **maximal matches** of a pattern?

Thanks for your attention!

References i

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

Bagan, G. (2006).
MSO queries on tree decomposable structures are computable with linear delay.
In *CSL*.

Florenzano, F., Riveros, C., Ugarte, M., Vansummeren, S., and Vrgoc, D. (2018).
Constant delay algorithms for regular document spanners.
In *PODS*.

References ii

Kazana, W. and Segoufin, L. (2013).
Enumeration of monadic second-order queries on trees.
TOCL, 14(4).

Losemann, K. and Martens, W. (2014).
MSO queries on trees: Enumerating answers under updates.
In *CSL-LICS*.

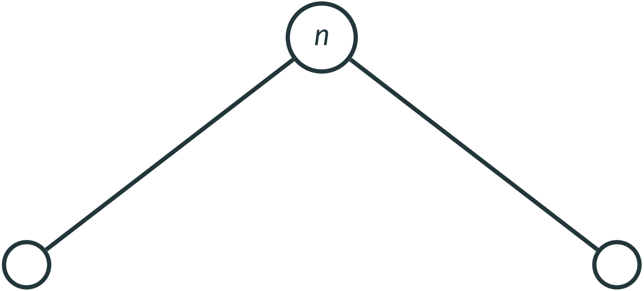
Niewerth, M. and Segoufin, L. (2018).
Enumeration of MSO queries on strings with constant delay and logarithmic updates.
In *PODS*.
To appear.

Proof idea for trees: set circuit construction (details)

- **Automaton:** *“Select all node pairs* (α, β)*”*
- **States:** $\{\emptyset, \alpha, \beta, \alpha\beta\}$

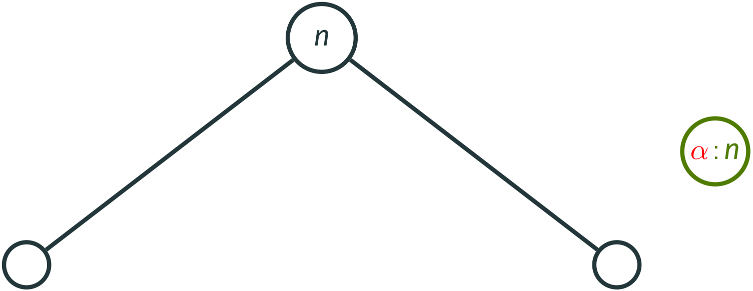
Proof idea for trees: set circuit construction (details)

- Automaton: *"Select all node pairs* (α, β)*"*
- States: $\{\emptyset, \alpha, \beta, \alpha\beta\}$



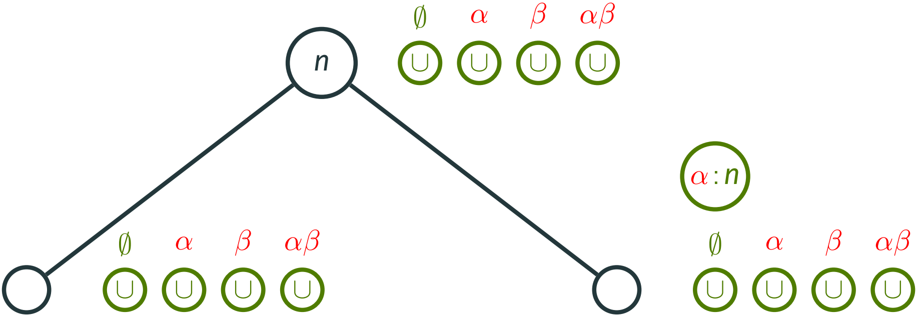
Proof idea for trees: set circuit construction (details)

- **Automaton:** *"Select all node pairs* (α, β)*"*
- **States:** $\{\emptyset, \alpha, \beta, \alpha\beta\}$



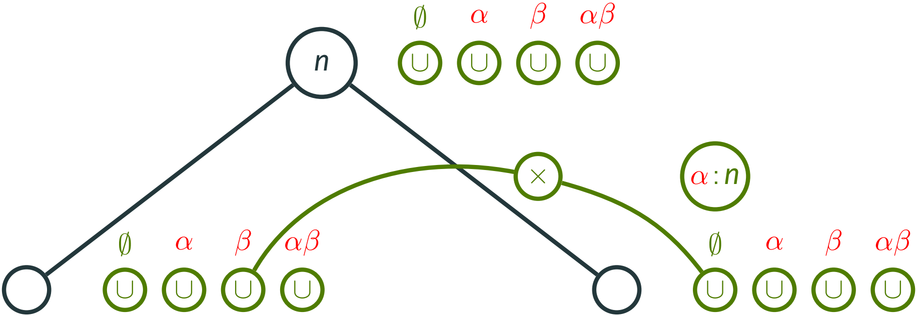
Proof idea for trees: set circuit construction (details)

- **Automaton:** *"Select all node pairs* (α, β)*"*
- **States:** $\{\emptyset, \alpha, \beta, \alpha\beta\}$



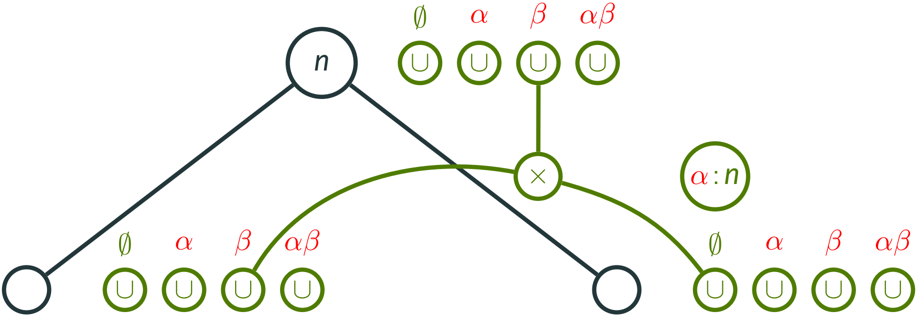
Proof idea for trees: set circuit construction (details)

- **Automaton:** *"Select all node pairs* (α, β)*"*
- **States:** $\{\emptyset, \alpha, \beta, \alpha\beta\}$



Proof idea for trees: set circuit construction (details)

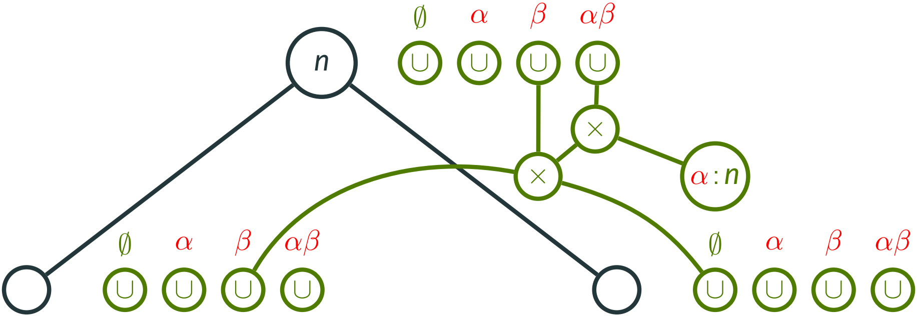
- **Automaton:** *“Select all node pairs* (α, β)*”*
- **States:** $\{\emptyset, \alpha, \beta, \alpha\beta\}$



n
$\emptyset$ α β $\alpha\beta$
$\cup$ $\cup$ $\cup$ $\cup$
$\times$
$\alpha : n$
$\emptyset$ α β $\alpha\beta$
$\cup$ $\cup$ $\cup$ $\cup$
$\emptyset$ α β $\alpha\beta$
$\cup$ $\cup$ $\cup$ $\cup$

Proof idea for trees: set circuit construction (details)

- **Automaton:** *“Select all node pairs* (α, β)*”*
- **States:** $\{\emptyset, \alpha, \beta, \alpha\beta\}$



Proof idea for trees: general enumeration approach

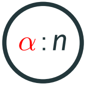
$\rightarrow$ Enumerate the set $T(g)$ captured by each gate g

$\rightarrow$ Do it by top-down induction on the circuit

Proof idea for trees: general enumeration approach

→ Enumerate the set $T(g)$ captured by each gate g

→ Do it by top-down induction on the circuit

Base case: variable 

:

Proof idea for trees: general enumeration approach

→ Enumerate the set $T(g)$ captured by each gate g

→ Do it by top-down induction on the circuit

Base case: variable $\alpha : n$: enumerate $\langle \alpha : n \rangle$ and stop

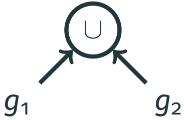
Proof idea for trees: general enumeration approach

→ Enumerate the set $T(g)$ captured by each gate g

→ Do it by **top-down induction** on the circuit

Base case: variable $\alpha : n$: enumerate $\langle \alpha : n \rangle$ and stop

∪-gate



Concatenation: enumerate $T(g_1)$
and then enumerate $T(g_2)$

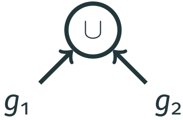
Proof idea for trees: general enumeration approach

→ Enumerate the set $T(g)$ captured by each gate g

→ Do it by **top-down induction** on the circuit

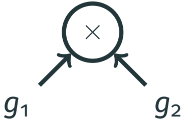
Base case: variable : enumerate $\langle \alpha : n \rangle$ and stop

$\cup$-gate



Concatenation: enumerate $T(g_1)$ and then enumerate $T(g_2)$

$\times$-gate

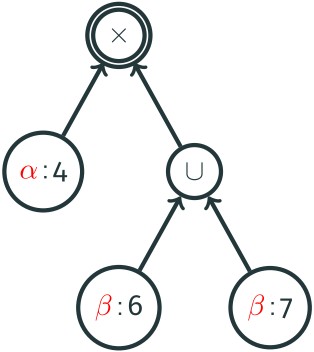


Lexicographic product:

```
for every $t_1$ in $T(g_1)$:
    for every $t_2$ in $T(g_2)$:
        output $t_1 + t_2$
```

Proof idea for trees: circuit conditions

Enumeration relies on some **conditions** on the input circuit (d-DNNF):



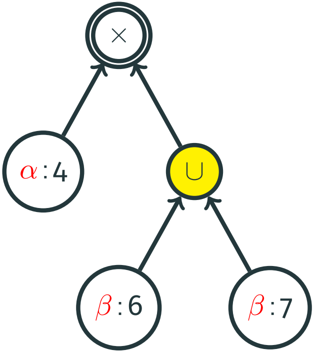
Proof idea for trees: circuit conditions

Enumeration relies on some conditions on the input circuit (d-DNNF):

- $\cup$ are all deterministic:

For any two inputs g_1 and g_2 of a $\cup$-gate, the captured sets $T(g_1)$ and $T(g_2)$ are disjoint (they have no tuple in common)

→ Avoids duplicate tuples



Proof idea for trees: circuit conditions

Enumeration relies on some **conditions** on the input circuit (d-DNNF):

- $\cup$ are all **deterministic**:

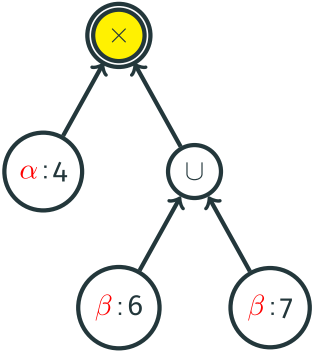
For any two inputs g_1 and g_2 of a $\cup$-gate, the captured sets $T(g_1)$ and $T(g_2)$ are **disjoint** (they have no tuple in common)

$\rightarrow$ Avoids **duplicate tuples**

- $\times$ are all **decomposable**:

For any two inputs g_1 and g_2 of a $\times$-gate, no variable has a path to both g_1 and g_2

$\rightarrow$ Avoids **duplicate singletons**



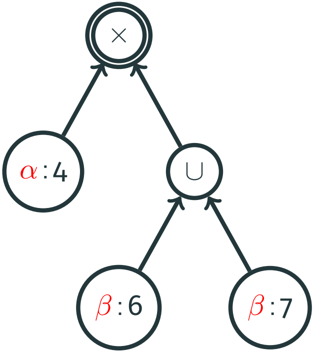
Proof idea for trees: circuit conditions

Enumeration relies on some conditions on the input circuit (d-DNNF):

- ∪ are all deterministic:

For any two inputs g_1 and g_2 of a ∪-gate, the captured sets $T(g_1)$ and $T(g_2)$ are disjoint (they have no tuple in common)

→ Avoids duplicate tuples



- × are all decomposable:

For any two inputs g_1 and g_2 of a ×-gate, no variable has a path to both g_1 and g_2

→ Avoids duplicate singletons

- Also an additional upwards-determinism condition

Proof idea for trees: circuit conditions

Enumeration relies on some **conditions** on the input circuit (d-DNNF):

- ∪ are all **deterministic**:

For any two inputs g_1 and g_2 of a ∪-gate, the captured sets $T(g_1)$ and $T(g_2)$ are **disjoint** (they have no tuple in common)

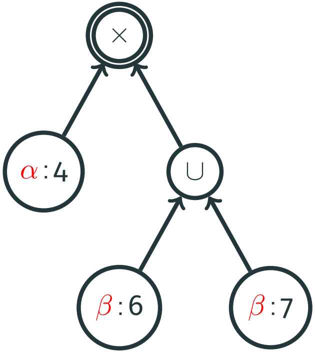
→ Avoids **duplicate tuples**

- × are all **decomposable**:

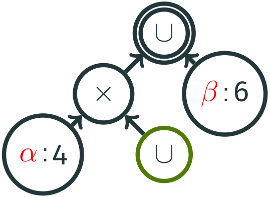
For any two inputs g_1 and g_2 of a ×-gate, no variable has a path to both g_1 and g_2

→ Avoids **duplicate singletons**

- Also an additional **upwards-determinism** condition
- Our circuit satisfies these thanks to **automaton determinism**

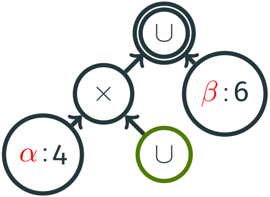


Proof idea for trees: enumeration subtleties



- We must not waste time in gates capturing $\emptyset$

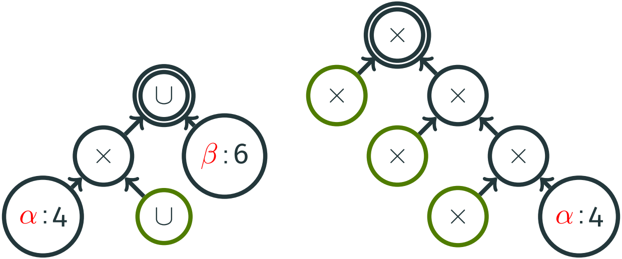
Proof idea for trees: enumeration subtleties



- We must not waste time in gates capturing $\emptyset$

  → **Label** them during the preprocessing

Proof idea for trees: enumeration subtleties

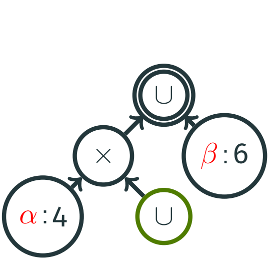


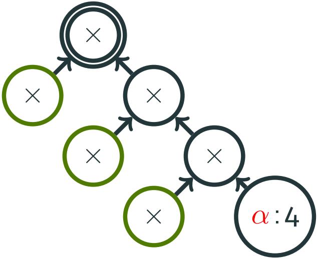
- We must not waste time in gates capturing $\emptyset$

  $\rightarrow$ **Label** them during the preprocessing

- We must not waste time because of gates capturing $\{\langle\rangle\}$

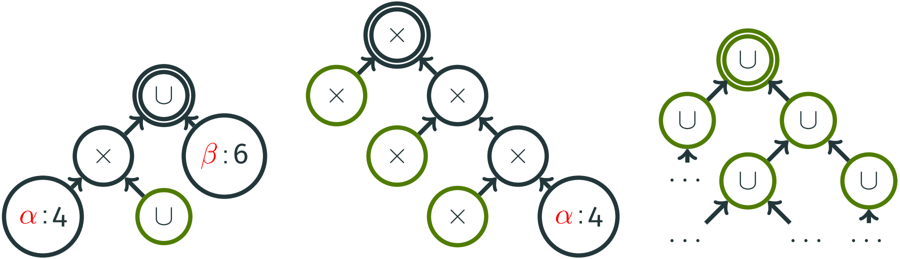
Proof idea for trees: enumeration subtleties





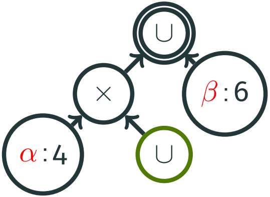
- We must not waste time in gates capturing $\emptyset$
  - $\rightarrow$ **Label** them during the preprocessing
- We must not waste time because of gates capturing $\{\langle\rangle\}$
  - $\rightarrow$ **Homogenization** to set them aside

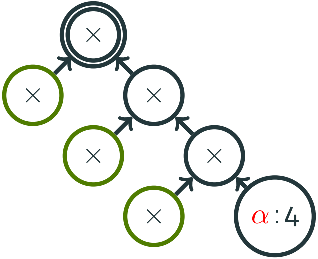
Proof idea for trees: enumeration subtleties

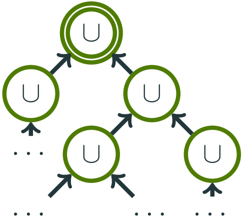


- We must not waste time in gates capturing $\emptyset$
  - $\rightarrow$ **Label** them during the preprocessing
- We must not waste time because of gates capturing $\{\langle\rangle\}$
  - $\rightarrow$ **Homogenization** to set them aside
- We must not waste time in **hierarchies of $\cup$-gates**

Proof idea for trees: enumeration subtleties







- We must not waste time in gates capturing $\emptyset$
  - → **Label** them during the preprocessing
- We must not waste time because of gates capturing $\{\langle\rangle\}$
  - → **Homogenization** to set them aside
- We must not waste time in **hierarchies of $\cup$-gates**
  - → Precompute a **reachability index** (uses **upwards-determinism**)