Regular languages recognition in restricted models of computation

Gabriel Bathie, Tatiana Starikovskaya

Definition

A regular language is a language described by a **regular expression**

- $\varnothing$ (empty set)
- ε (empty word)
- Letter a in an alphabet Σ

base case

If e_1, e_2 are regular expressions, then

- Union $(e_1 \mid e_2)$ is a regular expression
- Concatenation $(e_1 \cdot e_2)$ is a regular expression
- Kleene's star: $(e_1)^*$ is a regular expression

induction

©matthewleadbeater

Definition

A regular language is a language described by a **regular expression**

- $L(\varnothing) = \varnothing$ (empty language)
- $L(\varepsilon) = \varepsilon$ (empty word)
- $L(a) = \{a\}$ for letter in an alphabet Σ

base case

If e_1, e_2 are regular expressions, then

- $L(e_1 \mid e_2) = L(e_1) \cup L(e_2)$
- $L(e_1 \cdot e_2) = \{u \in \Sigma^* \mid u = vw, v \in L(e_1), w \in L(e_2)\}$
- $L((e_1)^*) = \{u_1 u_2 \ldots u_n \mid n \geq 0, u_i \in L(e_1)\}$

induction

©matthewleadbeater

Definition

A regular language is a language described by a **regular expression**

regular expression $(b)^*\ (ab \mid b)\ ab$

regular language $\{abab, babab, bbabab, bbbabab, \ldots\} \cup \{bab, bbab, bbbab, bbbbab, \ldots\}$

we will also use shorthands $e^+ := ee^*$**,**

$S^* := \{a_1 \mid a_2 \mid \ldots \mid a_m\}$ **for a set** $S = \{a_1, a_2, \ldots, a_m\}$

$[\hat{}a_i] := \{a_1 \mid a_2 \mid \ldots \mid a_{i-1} \mid a_{i+1} \mid \ldots \mid a_m\}$ **for an alphabet** $\{a_1, a_2, \ldots, a_m\}$

Regular language recognition

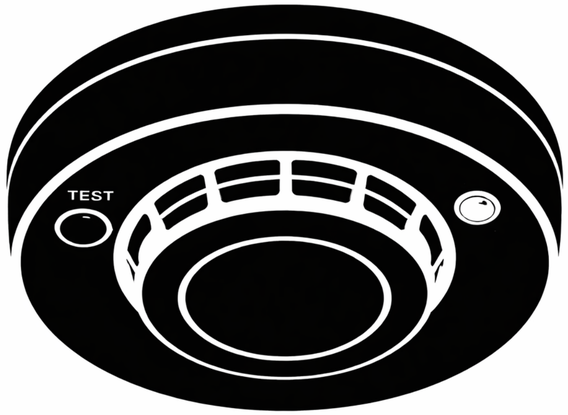
Input: an input S of length n and a regular language L

Output: Does $S \in L$?

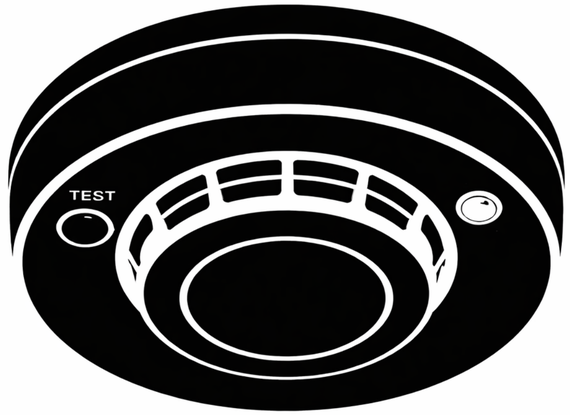
Important primitive in many applications:

- Event detection
- Protein search
- XML queries
- etc

Event detection



heat



smoke

hSShHsS ShhhhSssH...

If there both heat (H) and smoke (S) are detected within n last minutes, there is fire!

Is the word formed by the last n letters matches $\Sigma^* \ [(H\Sigma^*S) \mid (S\Sigma^*H)] \ \Sigma^*$?

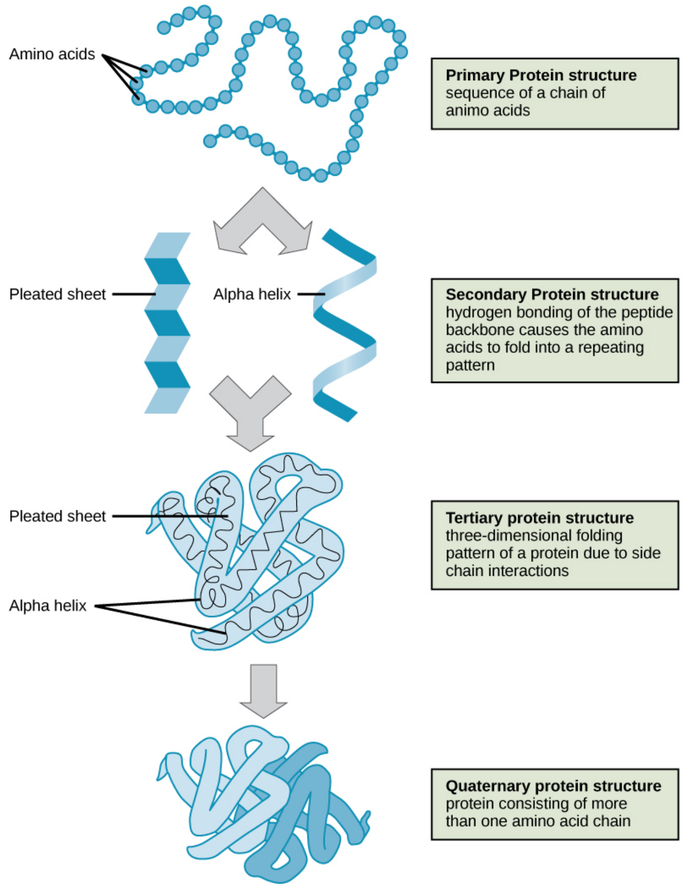
(here Σ^* stands for $(s \mid h \mid S \mid H)^*$)

Protein search

- A well-known protein motif is the N-glycosylation site, defined by the pattern:

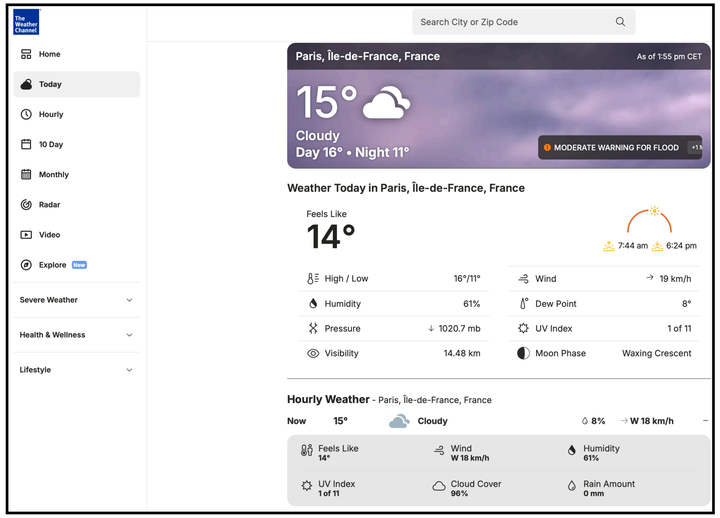
N[^P][ST][^P]

- N-glycosylation sites are spots where sugar molecules can attach to a protein
- Those sugars can change how the protein folds, works, and how the body recognises it
- Essential for drug design



Proteins Copyright © 2012 by OpenStaxCollege

XML queries



weather.com

```
<div data-testid="WeatherDetailsLabel" class="WeatherDetailsListItem--label--U+Wrx">High / Low</div>
<div data-testid="wxData" class="WeatherDetailsListItem--wxData--lW-7H">
    <span data-testid="TemperatureValue" dir="ltr">
        16
        <span>°</span>
        <span></span>
    </span>
    /
    <span data-testid="TemperatureValue" dir="ltr">
        11
        <span>°</span>
        <span></span>
    </span>
</div>
```

source code

To extract temperatures, find substrings matching **<span data-testid=“TemperatureValue”[∑*] **

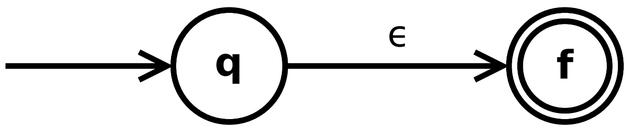
Outline of the rest of the talk

- Classical construction: Thompson automaton
- Time bounds
  - Improvements in time
  - Conditional time lower bounds
  - Parametrised algorithms
- Restricted models of computation
  - Streaming
  - Property testing (Gabriel Bathie)
  - Sliding window

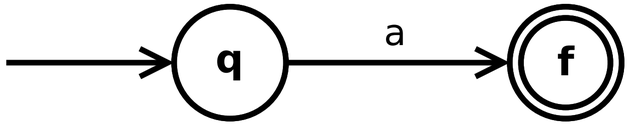
Part I
Classical construction: Thompson automaton

Thompson automaton

Thompson automaton is NFA defined inductively



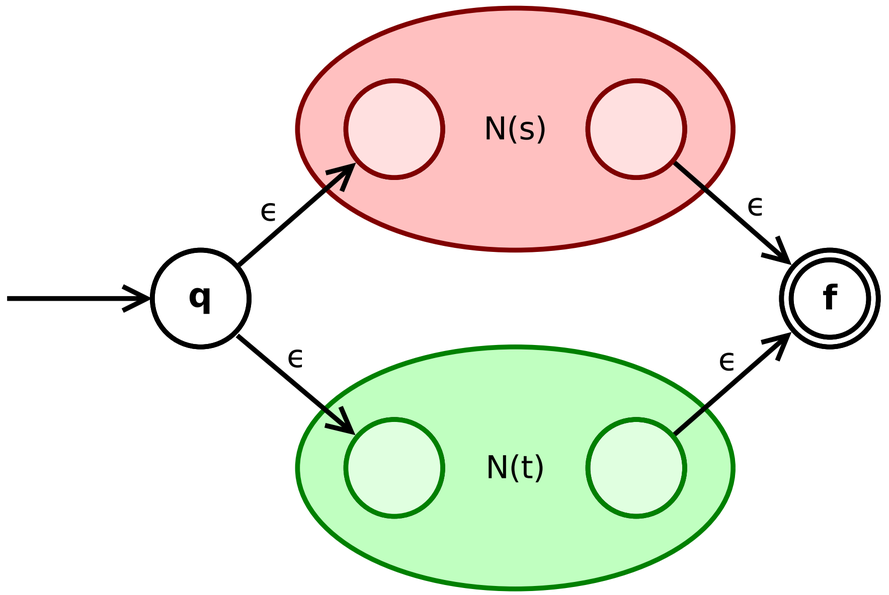
$N(\epsilon)$



$N(a)$

By Trapmoth - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=21861520

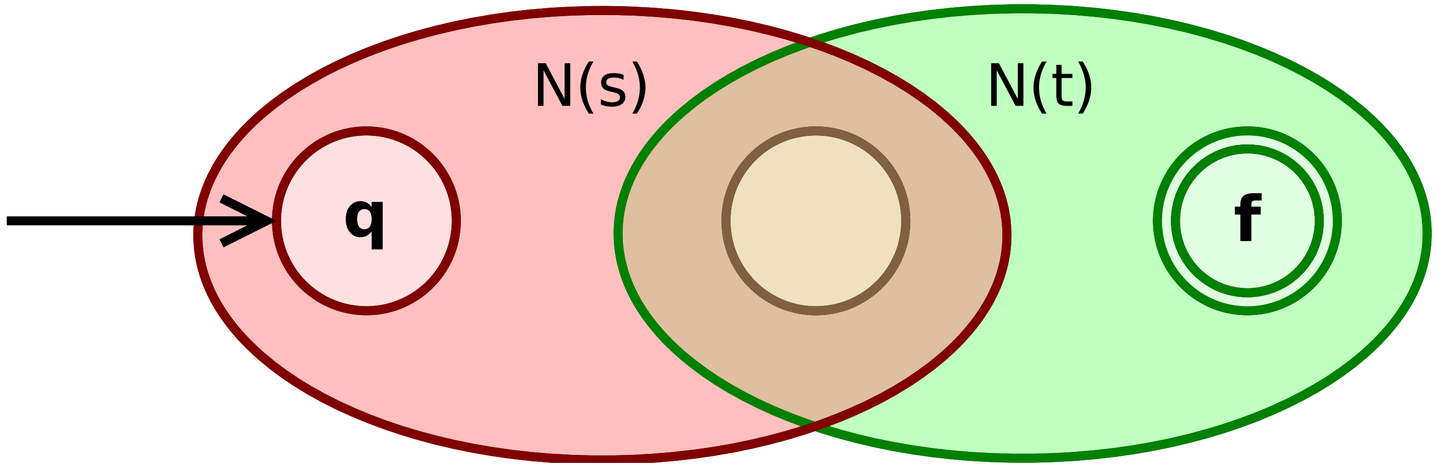
Thompson automaton



$N(s \mid t)$

By Trapmoth - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=21861520

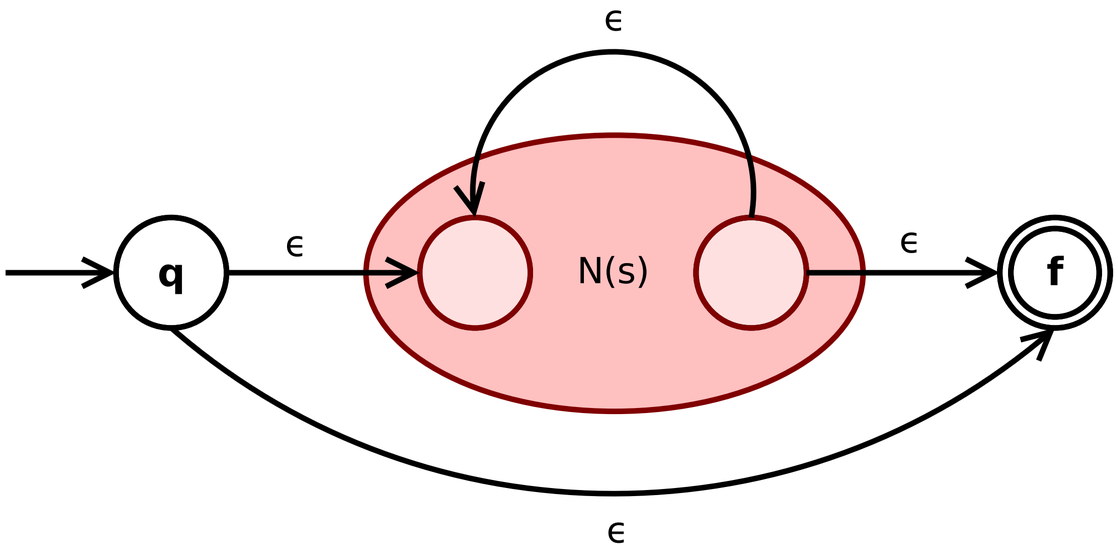
Thompson automaton



$N(s \cdot t)$

By Trapmoth - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=21861520

Thompson automaton

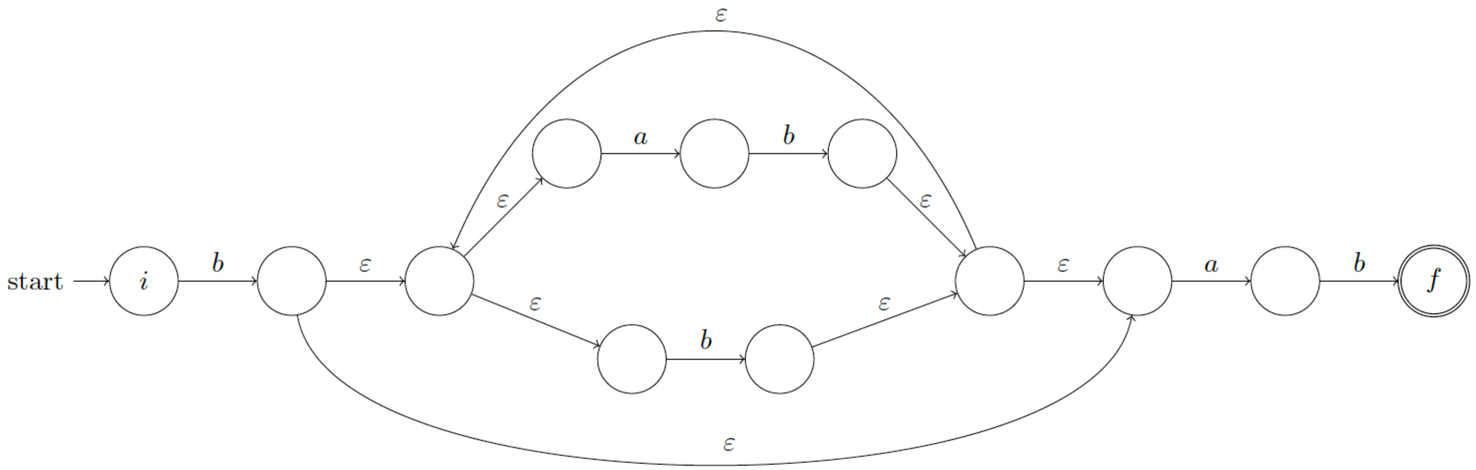


$N(s*)$

By Trapmoth - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=21861520

Membership in regular languages

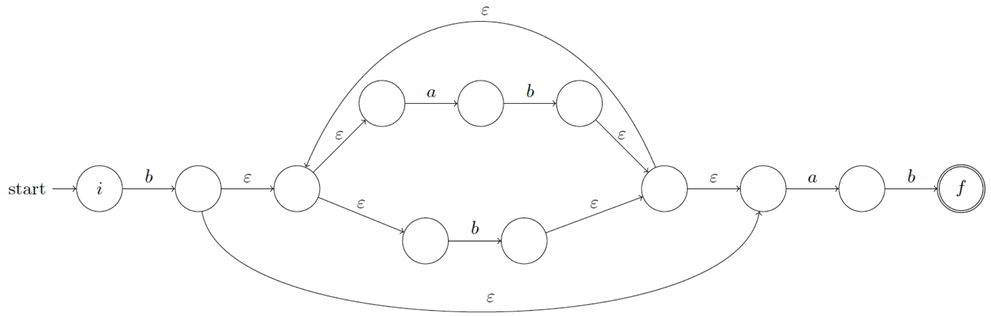
Thompson automaton of $b(ab\,|\,a)^*ab$



Size is $O(m)$, where m is the number of concatenations, unions, and Kleene stars
(every step increases size by constant)

Membership in regular languages

Thompson automaton of $b(ab \mid a)^*ab$ and a string $S = babab$

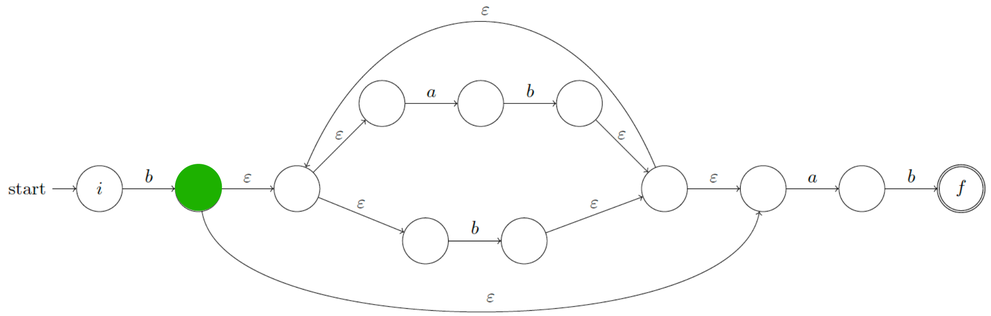


If a regular expression has size m, the problem can be solved in $O(mn)$ time and $O(m)$ space:

- Scan S from left to right
- Maintain the subset of the states accessible from the initial state with the current prefix of S
- If the final state can be reached with S, it is in the language

Membership in regular languages

Thompson automaton of $b(ab \mid a)^*ab$ and a string $S = babab$

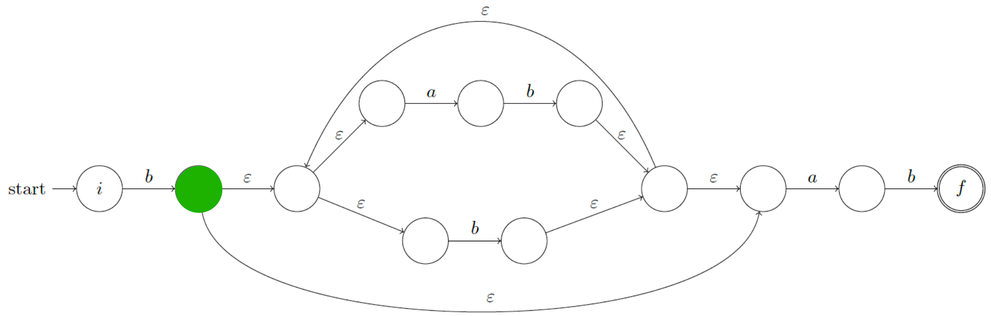


If a regular expression has size m, the problem can be solved in $O(mn)$ time and $O(m)$ space:

- Scan S from left to right
- Maintain the subset of the states accessible from the initial state with the current prefix of S
- If the final state can be reached with S, it is in the language

Membership in regular languages

Thompson automaton of $b(ab \mid a)*ab$ and a string $S = babab$

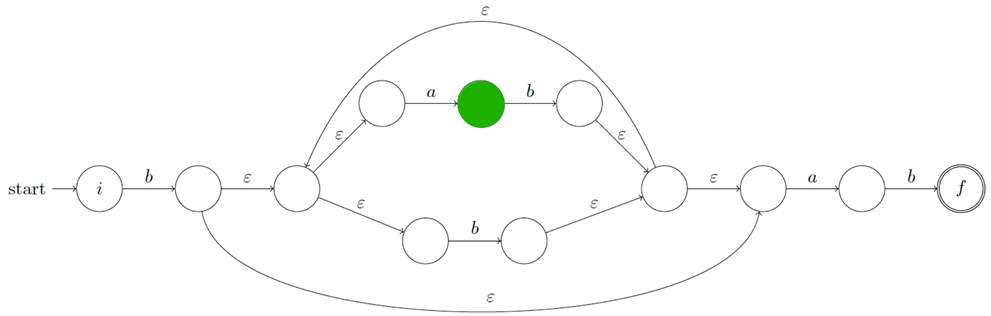


If a regular expression has size m, the problem can be solved in $O(mn)$ time and $O(m)$ space:

- Scan S from left to right
- Maintain the subset of the states accessible from the initial state with the current prefix of S
- If the final state can be reached with S, it is in the language

Membership in regular languages

Thompson automaton of $b(ab \mid a)^*ab$ and a string $S = babab$

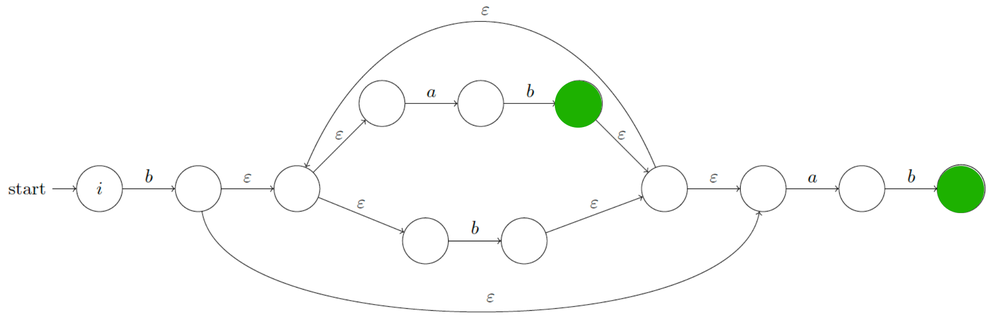


If a regular expression has size m, the problem can be solved in $O(mn)$ time and $O(m)$ space:

- Scan S from left to right
- Maintain the subset of the states accessible from the initial state with the current prefix of S
- If the final state can be reached with S, it is in the language

Membership in regular languages

Thompson automaton of $b(ab \mid a)^*ab$ and a string $S = babab$

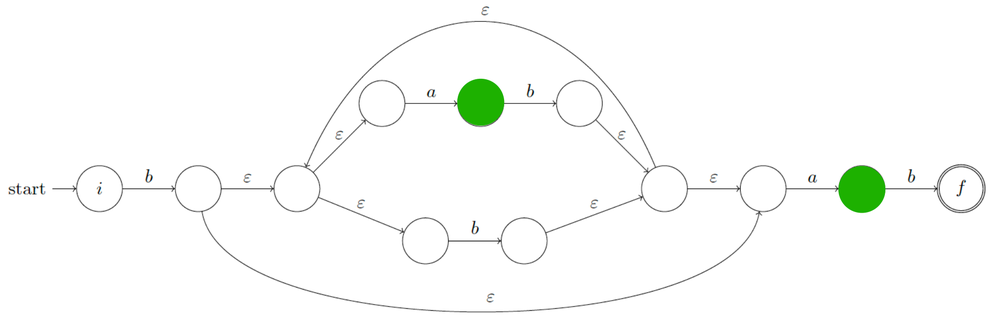


If a regular expression has size m, the problem can be solved in $O(mn)$ time and $O(m)$ space:

- Scan S from left to right
- Maintain the subset of the states accessible from the initial state with the current prefix of S
- If the final state can be reached with S, it is in the language

Membership in regular languages

Thompson automaton of $b(ab \mid a)^*ab$ and a string $S = babab$

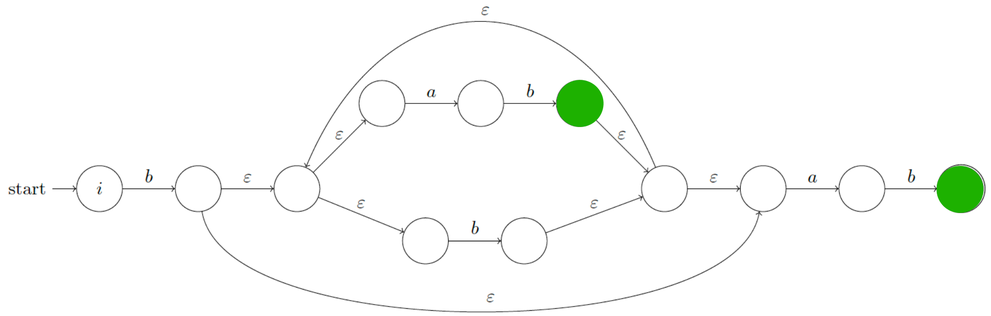


If a regular expression has size m, the problem can be solved in $O(mn)$ time and $O(m)$ space:

- Scan S from left to right
- Maintain the subset of the states accessible from the initial state with the current prefix of S
- If the final state can be reached with S, it is in the language

Membership in regular languages

Thompson automaton of $b(ab \mid a)^*ab$ and a string $S = babab$



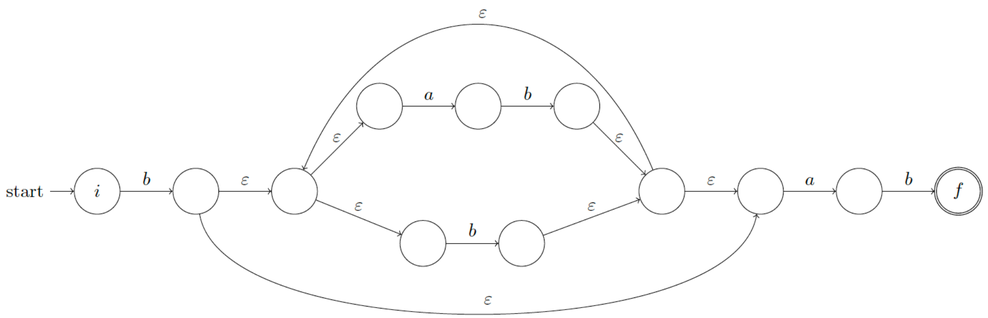
If a regular expression has size m, the problem can be solved in $O(mn)$ time and $O(m)$ space:

- Scan S from left to right
- Maintain the subset of the states accessible from the initial state with the current prefix of S
- If the final state can be reached with S, it is in the language

Membership in regular languages

Input: an input string S of length n and a regular language L

Output: Does $S \in L$?

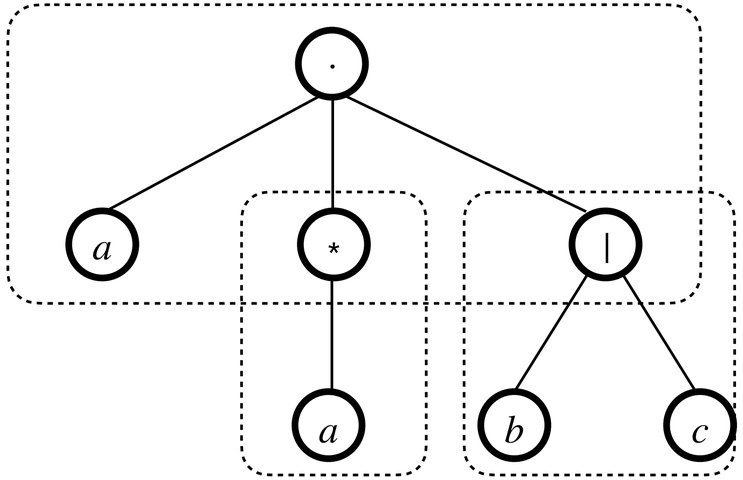


As the automaton is nondeterministic, checking if it accepts S seems to require $\Omega(m)$ time per letter (or $\Omega(mn)$ in total) and $\Omega(m)$ space

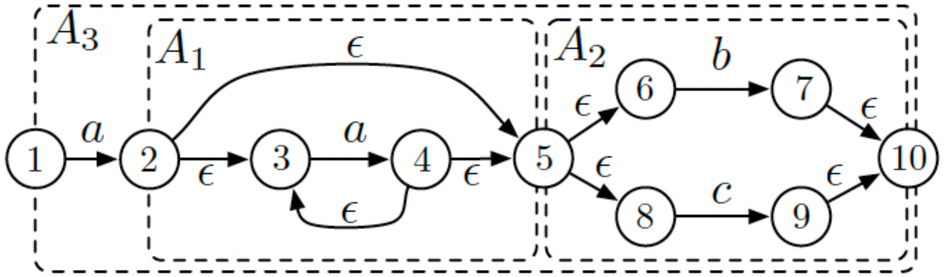
Part II
Time bounds

Are there faster algorithms?

Polylog improvements



Parse tree of $a(a)*(b\,|\,c)$

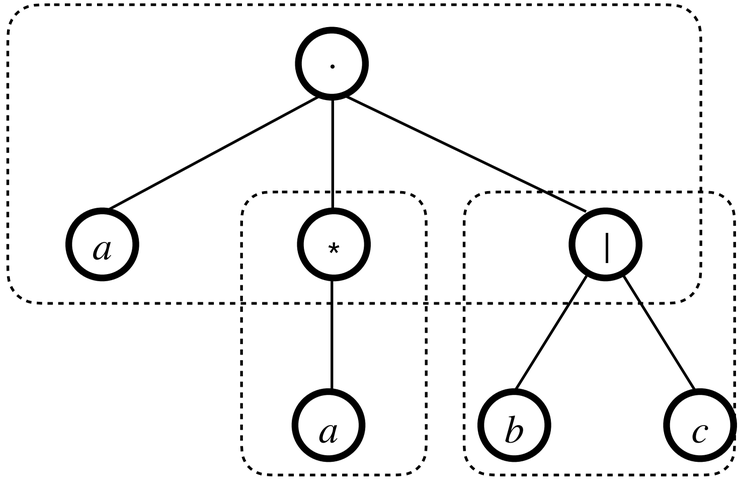


Thompson automaton of $a(a)*(b\,|\,c)$

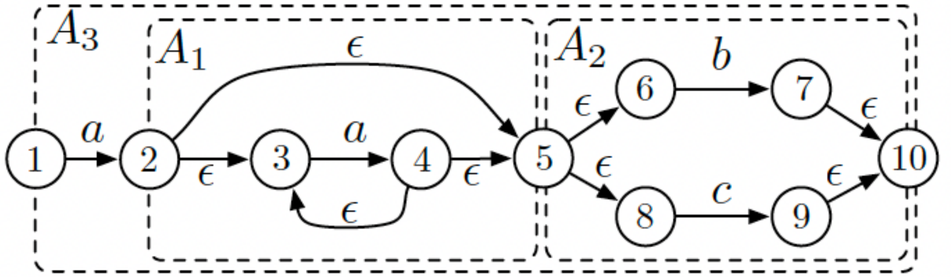
Image credit: Bille, Thorup "Faster Regular Expression Matching"

Myers [JACM'92]: $O(\frac{nm}{\log n} + (n+m)\cdot \log n)$-time algorithm

Polylog improvements



Parse tree of $a(a)*(b\,|\,c)$

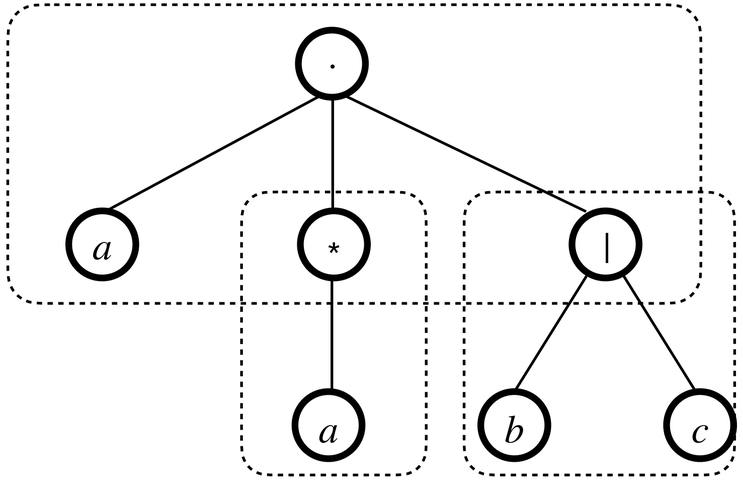


Thompson automaton of $a(a)*(b\,|\,c)$

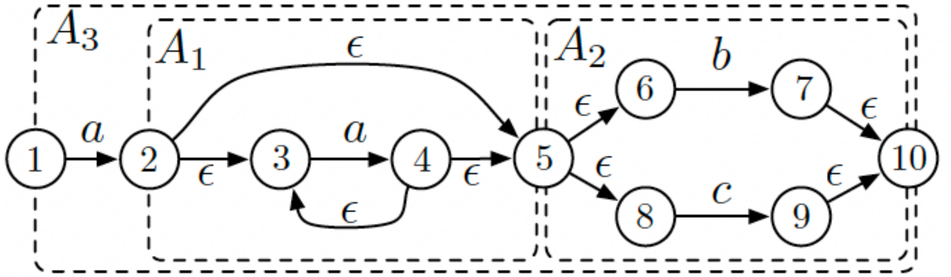
Image credit: Bille, Thorup “Faster Regular Expression Matching”

- Decompose the parse tree into $O(m/x)$ subtrees of size x
- Infer a decomposition of the Thompson automaton into $O(m/x)$ sub-automata

Polylog improvements



Parse tree of $a(a)*(b\,|\,c)$

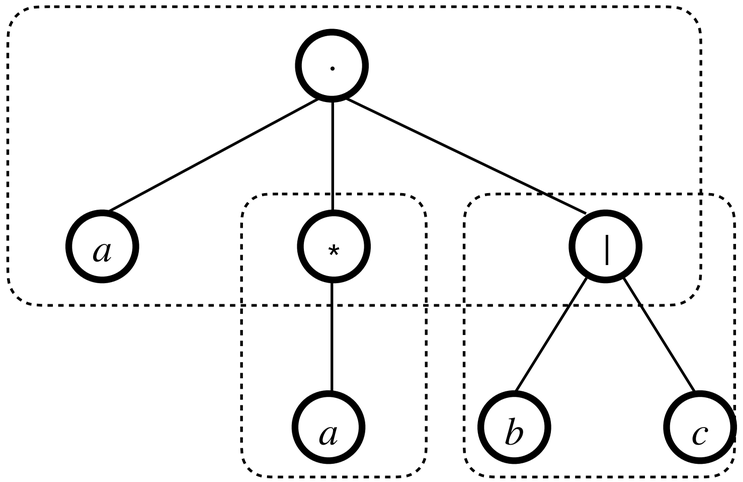


Thompson automaton of $a(a)*(b\,|\,c)$

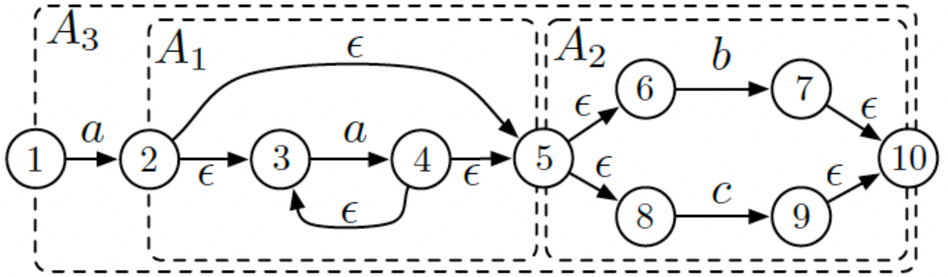
Image credit: Bille, Thorup “Faster Regular Expression Matching”

- Sub-automata can be encoded via the corresponding subtrees using $O(x)$ bits
- If $x = o(\log m)$, many sub-automata will have the same code

Polylog improvements



Parse tree of $a(a)*(b\,|\,c)$



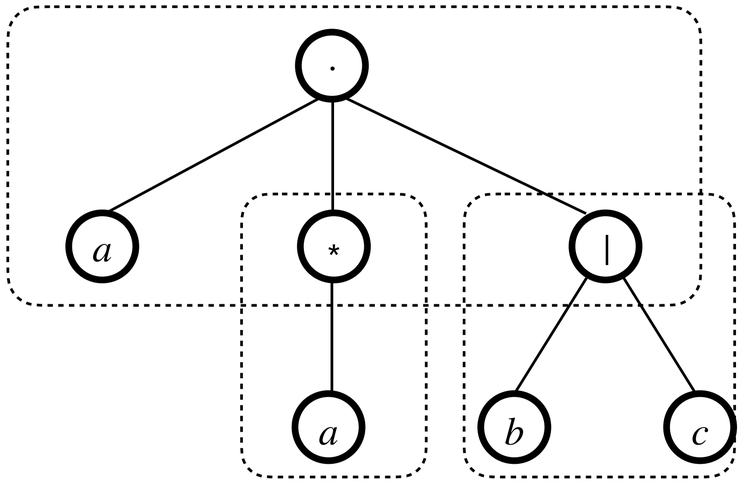
Thompson automaton of $a(a)*(b\,|\,c)$

Image credit: Bille, Thorup "Faster Regular Expression Matching"

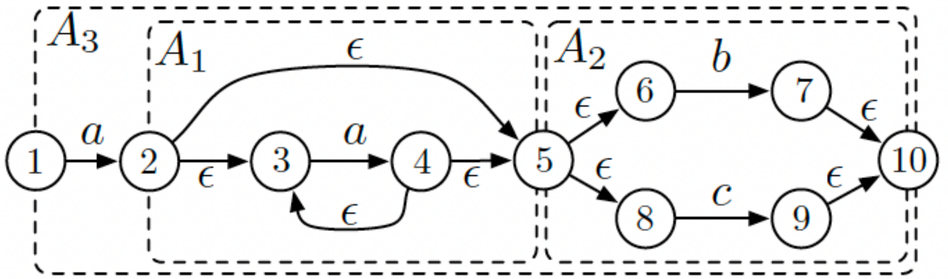
- Create a universal table T of size $2^{O(x)}$
- For each sub-automaton A, a local set of states U_A, and $\alpha \in \Sigma \cup \{\varepsilon\}$

$T[A, U_A, \alpha] = \delta(U_A, \alpha) :=$ **set of states reachable from U_A by an edge labelled α**

Polylog improvements



Parse tree of $a(a)*(b\,|\,c)$

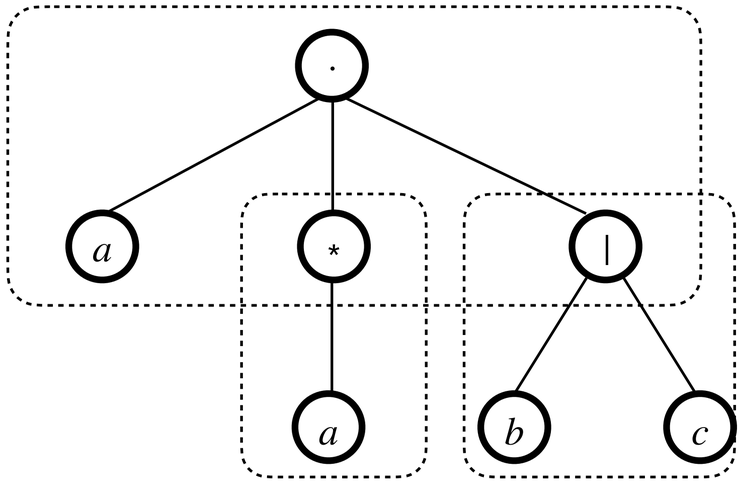


Thompson automaton of $a(a)*(b\,|\,c)$

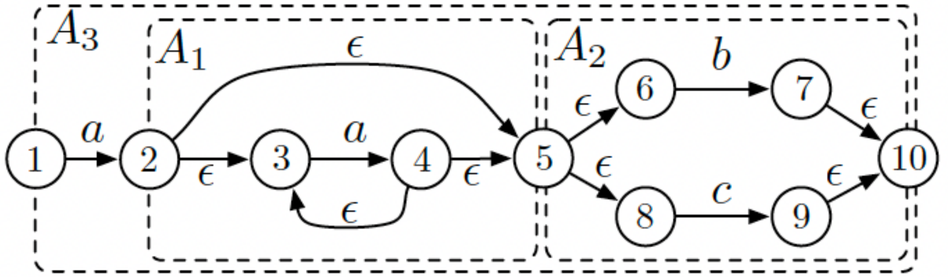
Image credit: Bille, Thorup "Faster Regular Expression Matching"

- **Naive algorithm:** start with U = {initial state}, update $U = \delta(U, \alpha)$ for each letter α of the input, accept if U contains a final state after reading the entire input
- Computing $\delta(U, \alpha)$: follow transitions labeled α from U ($O(m/x)$ time using the table T), and then traverse paths of ε-transitions **(difficulty: such paths can lead to distant sub-automata)**

Polylog improvements



Parse tree of $a(a)*(b\,|\,c)$



Thompson automaton of $a(a)*(b\,|\,c)$

Image credit: Bille, Thorup "Faster Regular Expression Matching"

- **Myers [JACM'92]:** any cycle-free path of ε-transitions in the Thompson automaton uses at most one of the back transitions we get from $*$
- **Corollary:** one can compute the result of following the ε-transitions via two depth-first traversals in $O(m/x)$ time

Polylog improvements (bit-packing + tabulation)

Myers [JACM'92]

- choose $x = 0.5 \log m$
- $O((nm)/\log n + (n+m) \cdot \log n)$-time and $O(m)$-space algorithm

Bille and Thorup [ICALP'09]

- similar sub-automata decomposition
- the table allows to advance multiple letters at a time
- $O((nm \log\log n)/\log^{1.5} n + n + m)$-time and $O(m)$-space algorithm

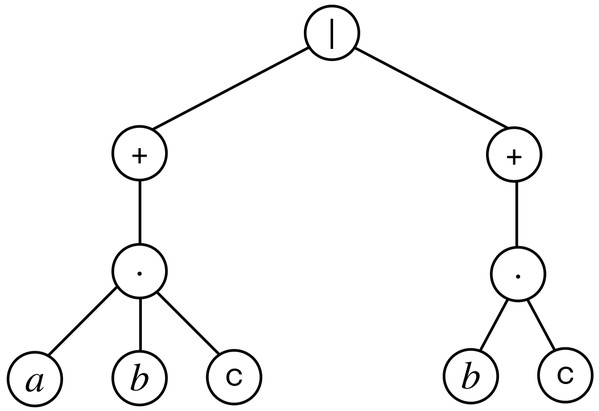
Lower bounds

Fine-grained complexity

Backurs and Indyk [FOCS'16]; Bringmann et al. [FOCS'17]

Classification of the so-called homogeneous regular expressions into "easy" and "hard", **excluding polynomial improvements** for the latter under SETH.

$$[abc]^+ \mid [bc]^+$$



Fine-grained complexity

Backurs and Indyk [FOCS'16]; Bringmann et al. [FOCS'17]

Classification of the so-called homogeneous regular expressions into "easy" and "hard", **excluding polynomial improvements** for the latter under SETH.

Type	Example	
$\cdot\vert\cdot$	$[a\vert bb][ba\vert b]$	
$\cdot\vert *$	$[a^*\vert b^*][c^*\vert b]$	
$\cdot\vert +$	$[a^+\vert b^+][c^+\vert b]$	
$\cdot+\cdot$	$[ab]^+[bca]^+$	
$\cdot+\vert$	$[a\vert b]^+[a\vert c\vert d]^+$	$\Omega((mn)^{1-\alpha})$
$\cdot+*$	$[a][a^+]^*[b^+]$	
$\cdot*\cdot$	$[ab]^*[bca]^*$	
$\cdot*\vert$	$[a\vert b]^*[a\vert b\vert c]^*$	
$\cdot*+$	$[a^*]b[b^+]^*$	

Type	Example	
$*\cdot\vert$	$[[a\vert b][b\vert c]]^*$	$O(n+m)$
$*\cdot*$	$[a^*b^*c^*]^*$	$\Omega((mn)^{1-\alpha})$
$*\cdot+$	$[a^+b^+c^+]^*$	
$*\vert\cdot$	$[a\vert ab\vert bc]^*$	
$*\vert*$	$[a^*\vert b^*\vert c^*]^*$	
$*\vert+$	$[a^+\vert b^+\vert c^+]^*$	$O(n+m)$
$*+\cdot$	$[[abcd]^+]^*$	
$*+\vert$	$[[a\vert b\vert c\vert d]^+]^*$	
$*+*$	$[[a^*]^+]^*$	

Fine-grained complexity

Backurs and Indyk [FOCS'16]; Bringmann et al. [FOCS'17]

Classification of the so-called homogeneous regular expressions into "easy" and "hard", **excluding polynomial improvements** for the latter under SETH.

Type	Example	
$+\cdot\|$	$[[a\|b][b\|c]]^+$	$O(n+m)$
$+\cdot*$	$[a^*b^*c^*]^+$	$\Omega((mn)^{1-\alpha})$
$+\cdot+$	$[a^+b^+c^+]^+$	$\tilde{\Theta}(nm^{1/3}+m)$
$+\|\cdot$	$[a\|ab\|bc]^+$	
$+\|*$	$[a^*\|b^*\|c^*]^+$	
$+\|+$	$[a^+\|b^+\|c^+]^+$	$O(n+m)$
$+*\cdot$	$[[abcd]^*]^+$	
$+*\|$	$[[a\|b\|c\|d]^*]^+$	
$+*+$	$[[a^+]^*]^+$	

Type	Example	
$\|\cdot\|$	$[(a\|b)(b\|c)]\|[(a\|c)b]$	$O(n+m)$
$\|\cdot*$	$[a^*b^*]\|[b^*c^*]$	$\Omega((mn)^{1-\alpha})$
$\|\cdot+$	$[a^+b^+]\|[b^+c^+]$	
$\|*\cdot$	$[abc]^*\|[bc]^*$	
$\|*\|$	$[a\|b\|c]^*\|[b\|c]^*$	
$\|*+$	$[a^+]^*\|[b^+]^*$	$O(n+m)$
$\|+\cdot$	$[abc]^+\|[bc]^+$	
$\|+\|$	$[a\|b\|c]^+\|[b\|c]^+$	
$\|+*$	$[a^*]^+\|[b^*]^+$	

Hardness for $\mid \circ *$

Orthogonal Vectors

- A set $A \subseteq \{0,1\}^d$ of N vectors, a set $B \subseteq \{0,1\}^d$ of N vectors
- Are there vectors $a \in A, b \in B$ such that $a \cdot b = 0$?

Lower bound for Orthogonal Vectors

For $d = \text{polylog } n$ and any constant $\alpha \in [0,1)$, there is no $O(N^{2-\alpha} \cdot d^{O(1)})$-time algorithm for the orthogonal vectors (assuming SETH)

Observation

It is enough to show that $\circ *$ is hard

Hardness for $\circ$ $*$

High-level idea

- $A \subseteq \{0,1\}^d \rightarrow$ a regular expression $R = [\ \circ_{i=1}^{i=|S|} (x*y*)]\ R'\ [\ \circ_{i=1}^{i=|S|} (x*y*)]$ of size $O(Nd)$
- $B \subseteq \{0,1\}^d \rightarrow$ a string S of size $O(Nd)$

- If exist $a \in A$, $b \in B$ with $a \cdot b = 0$, then a substring of S can be derived from R'
- Otherwise, **no substring** of S can be derived from R'
- Construction time is $O(Nd)$

Hardness for $\circ^*$

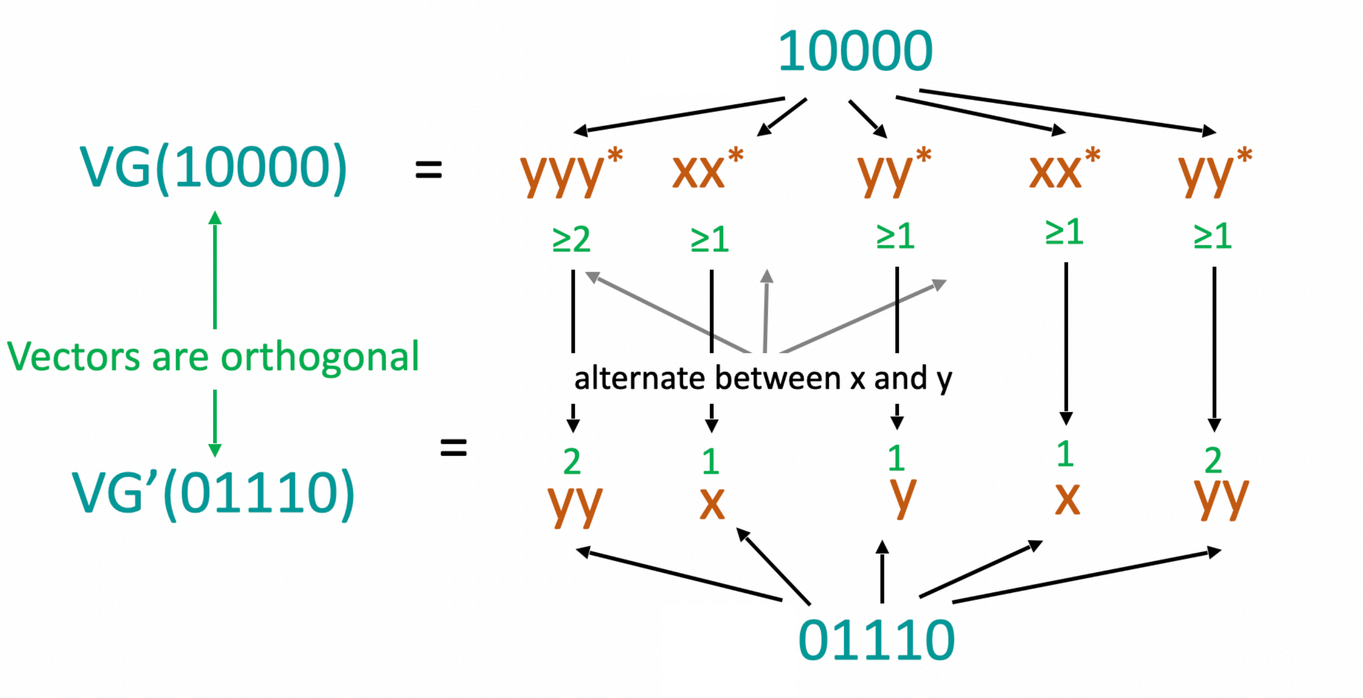
Vector gadgets

- $a \in A \rightarrow$ regexp $VG(a)$ $\quad\quad |VG(b)| = O(d)$
- $b \in B \rightarrow$ string $VG'(b)$ $\quad\quad |VG'(a)| = O(d)$

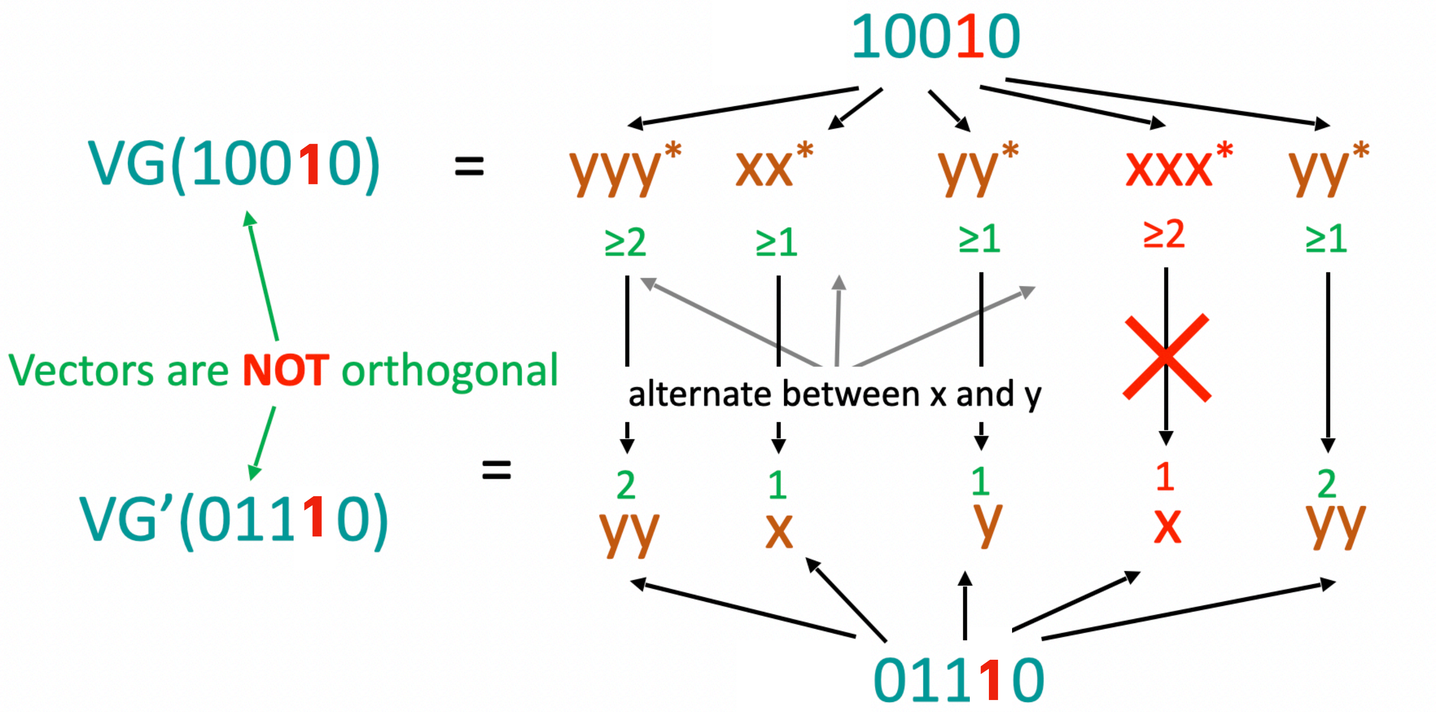
- If a and b are **orthogonal**, then $VG'(b)$ **can** be derived from $VG(a)$
- If a and b are **NOT orthogonal**, then $VG'(b)$ **can't** be derived from $VG(a)$

Hardness for $\circ$ *

- $a \in A \rightarrow$ regexp $VG(a)$
- $b \in B \rightarrow$ string $VG'(b)$
- a and b are **orthogonal** iff $VG'(b)$ **can** be derived from $VG(a)$



Hardness for $\circ$ *



- $a \in A \rightarrow$ regexp $VG(a)$
- $b \in B \rightarrow$ string $VG'(b)$
- a and b are **orthogonal** iff $VG'(b)$ **can** be derived from $VG(a)$

Hardness for $\circ$ *

High-level idea

- $A \subseteq \{0,1\}^d \rightarrow$ a regular expression $R = [\circ_{i=1}^{i=|S|} (x*y*)]\ R'\ [\circ_{i=1}^{i=|S|} (x*y*)]$ of size $O(Nd)$
- $B \subseteq \{0,1\}^d \rightarrow$ a string S of size $O(Nd)$

- If exist $a \in A$, $b \in B$ with $a \cdot b = 0$, then a substring of S can be derived from R'
- Otherwise, **no substring** of S can be derived from R'

- Construct R': concatenate the vector gadgets $VG(a)$ for $a \in A$ with some extra padding
- Construct S: concatenate the vector gadgets $VG'(b)$ for $b \in B$ with some extra padding

- The construction time is $O(Nd)$

Even finer bounds

Abboud and Bringmann [ICALP'18]

An even more fine-grained approach, **excluding $O(nm/\log^{7+\varepsilon} n)$-time algorithm** under Formula-SAT Hypothesis

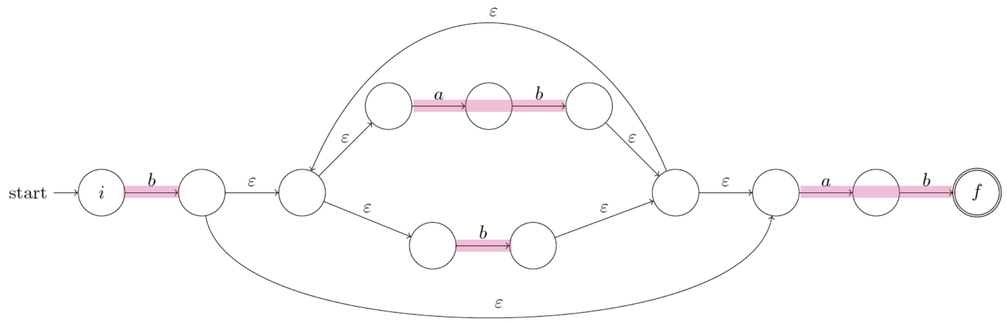
Schepper [ESA'20]

- $nm/2^{\Omega(\sqrt{\log n})}$-time algorithms for $|\circ|$ (example: $(a(a\,|\,b))\,|\,((a\,|\,b)(b\,|\,c))$) and $|\circ+$ (example: $(a^+b^+c^+)\,|\,(a^+b^+)$) via batched orthogonal vector queries
- $\Theta(nm/\log^c n)$ time bound for other "hard" homogeneous regular expressions assuming the formula-SAT hypothesis

Can we find good parameters?

Number of words

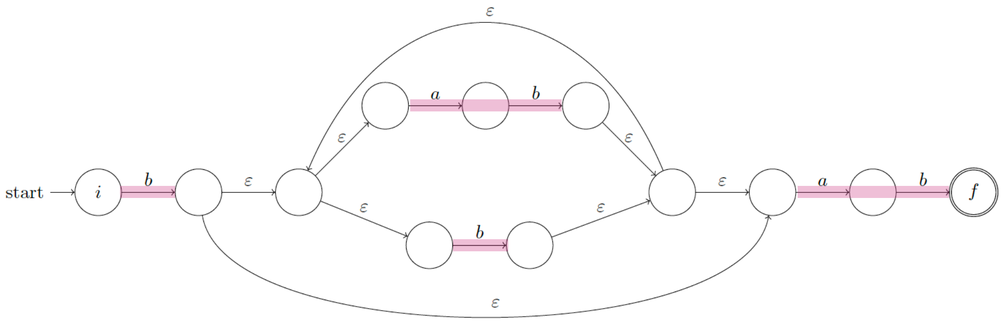
$$\underbrace{b}_{1}\underbrace{(ab}_{2}\,|\,\underbrace{b)}_{3}{}^{*}\underbrace{ab}_{4}$$



- d = **number of "words"** in a regular expression
- Usually **small** in practice

Number of words: efficient algorithms

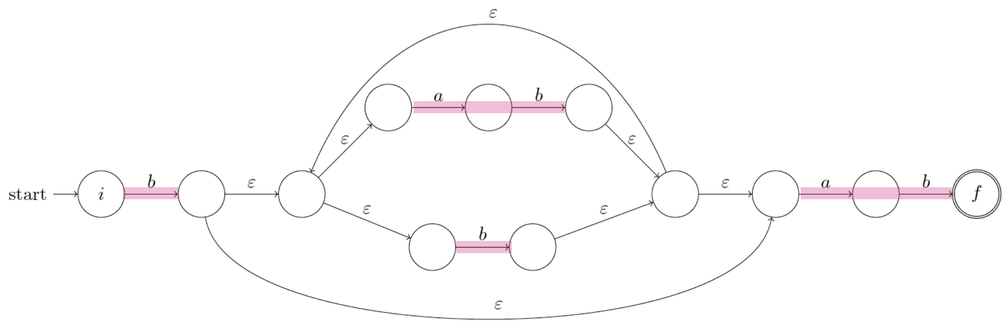
$$\underset{1}{b}(\underset{2}{ab}\,|\,\underset{3}{b})^*\underset{4}{ab}$$



Bille and Thorup [SODA'10]: regular languages membership can be solved online in $O(n\frac{d \log w}{w} + \log d)$ time and $\Theta(m)$ space

(here, $w = \Theta(\log n)$ is the size of the machine word)

High-level idea of Bille and Thorup [SODA'10]

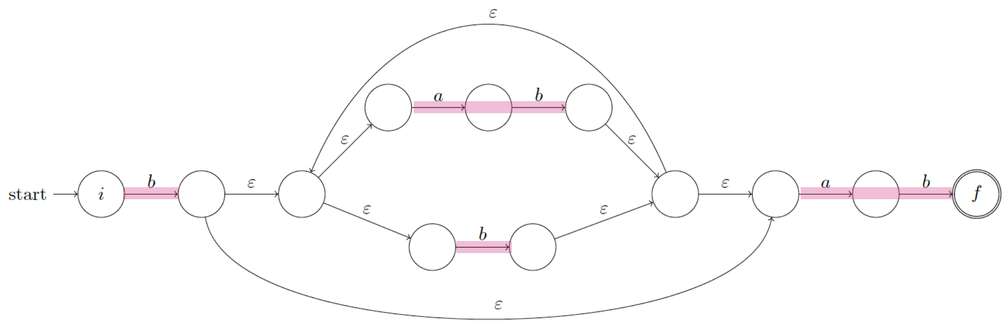


$S = babbbab$

$D[3, ab] = 1$

- **Dynamic programming table** D, indexed by positions of S and words in the regular expression
- $D[i, w] = 1$ iff there is a path in the automaton from the initial state to the end of the word w labelled with $S[1..i]$
- Run the **dictionary matching algorithm** for a dictionary of all "words" in R and a text S:
  - $O(m)$ space and $O(nd)$ time
  - For every i, reports all words w in the regular expressions such that $S[1..i]$ ends with w
- $D[i, w] = 1$ iff $S[1..i]$ ends with w and there is w' such that $D[i - |w|, w'] = 1$ and there is an ε-transition path from w' to w

High-level idea of Bille and Thorup [SODA'10]



$S = babbbab$

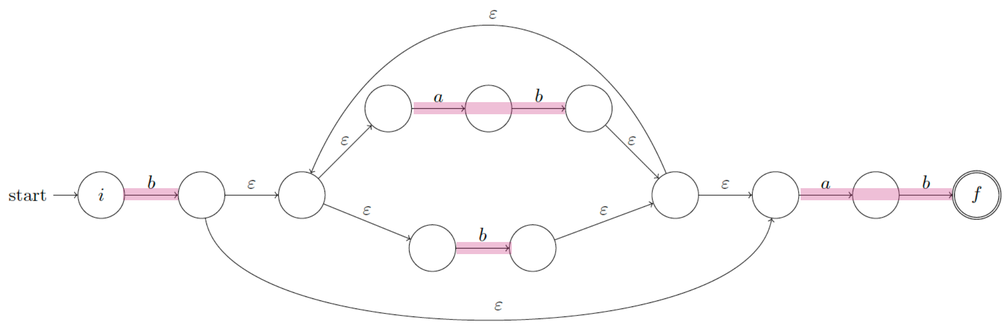
$D[3, ab] = 1$

$D[4, b] = 1$ because:

- $S[1,4] = babb$ ends with b
- $D[3, ab] = 1$
- There is an ε-transition path from ab to b

- **Dynamic programming table** D, indexed by positions of S and words in the regular expression
- $D[i, w] = 1$ iff there is a path in the automaton from the initial state to the end of the word w labelled with $S[1..i]$
- Run the **dictionary matching algorithm** for a dictionary of all "words" in R and a text S:
  - $O(m)$ space and $O(nd)$ time
  - For every i, reports all words w in the regular expressions such that $S[1..i]$ ends with w
- $D[i, w] = 1$ iff $S[1..i]$ ends with w and there is w' such that $D[i - |w|, w'] = 1$ and there is an ε-transition path from w' to w

High-level idea of Bille and Thorup [SODA'10]



$S = babbbab$

$D[3, ab] = 1$

More work required in order to achieve the final complexity... (main idea: bit-packing & tabulation)

- **Dynamic programming table** D, indexed by positions of S and words in the regular expression
- $D[i, w] = 1$ iff there is a path in the automaton from the initial state to the end of the word w labelled with $S[1..i]$
- Run the **dictionary matching algorithm** for a dictionary of all "words" in R and a text S:
  - $O(m)$ space and $O(nd)$ time
  - For every i, reports all words w in the regular expressions such that $S[1..i]$ ends with w
- $D[i, w] = 1$ iff $S[1..i]$ ends with w and there is w' such that $D[i - |w|, w'] = 1$ and there is an ε-transition path from w' to w

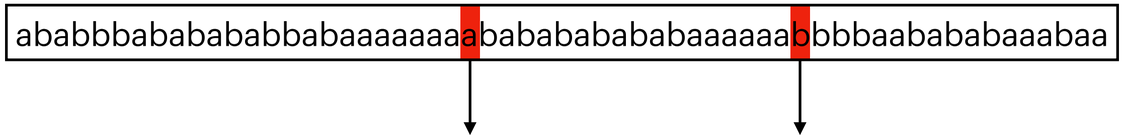
Conclusion for Part II

- For general regular expressions, achieving $\Theta((mn)^{1-\varepsilon})$-time algorithms for membership is impossible (conditional on SETH)
- **Fastest algorithm:** Bille & Thorup, ICALP'09 in $O(mn \log\log(n)/\log^{1.5} n + m + n)$ time
- Better bounds are known for certain types of homogeneous regular expressions
- $O(n\frac{d \log w}{w} + \log d)$-time algorithm (Bille & Thorup, SODA'10), where d is the number of words in a regular expression

Part III
Restricted-access settings

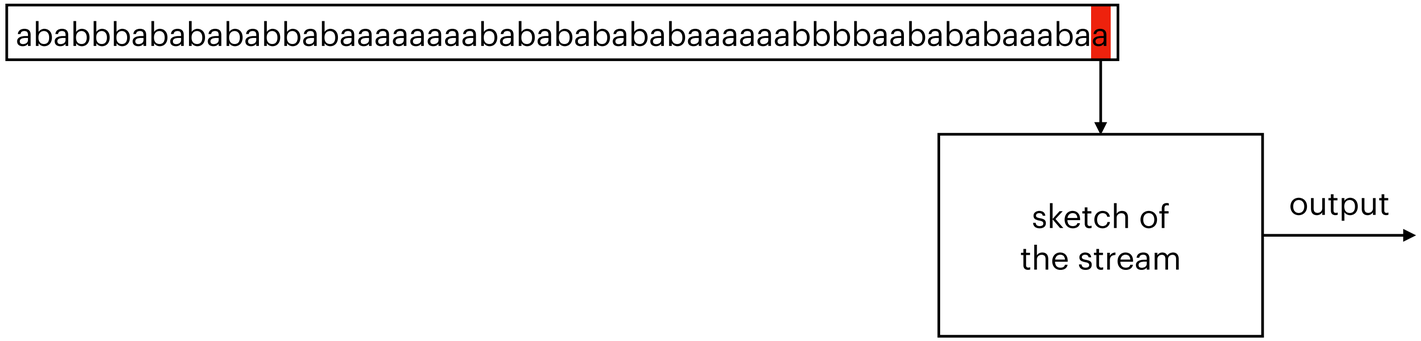
Definitions

Classical offline setting



- We store the whole input
- We can access any letter in $O(1)$ time

Streaming setting



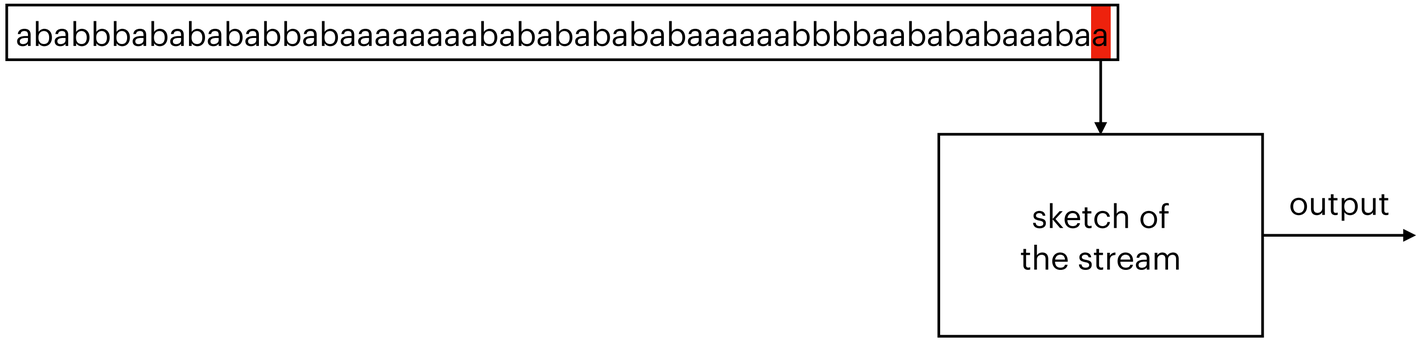
- Input arrives as a stream, one letter at a time
- When a new letter arrives, we update the sketches, and possibly answer a question about the stream
- We account for all the space used
- We can’t go back and read a letter we’ve seen previously (unless storing it)

Streaming setting

We are in the **standard word RAM model**

- space = number of machine words used (think of the number of integers stored)
- time = number of elementary operations (memory allocation, arithmetic operations, if/else, goto, ...)
- we care (mainly) about the asymptotic of these parameters

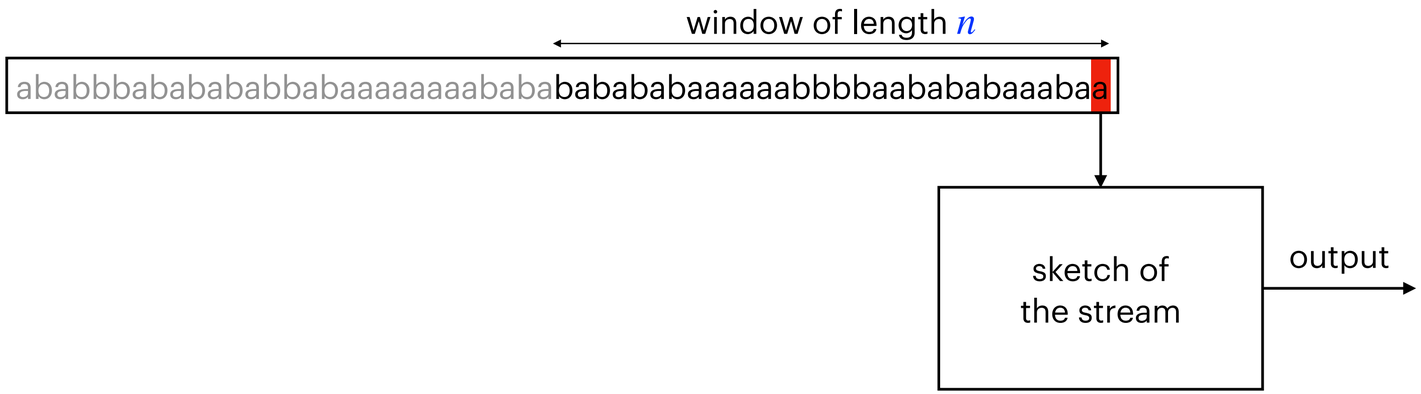
Streaming setting



Applications

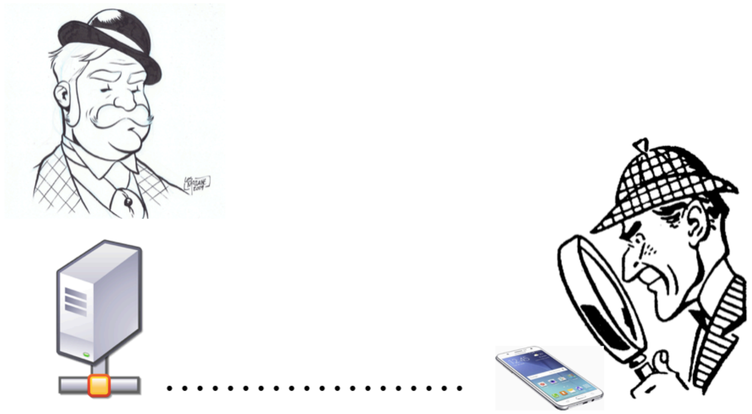
- Input is an output of another algorithm, we can’t afford to store it
- Input consists of sequential measurements

Sliding window setting



- Sliding window setting is streaming setting
- Questions can only be asked about the window of the last n letters
- Application: Input is sequential measurements that quickly become outdated

Property testing



- Input is stored on a server
- Accessing letters of the input is expensive
- **Task:** decide whether the input has a certain property
- **Query complexity:** number of letter accesses

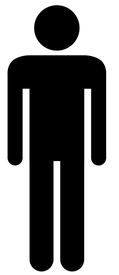
Property testing
Gabriel Bathie

Streaming setting

Lower bound

Set Disjointness

Alice holds
subset $S_1 \subseteq \{1,\ldots,m\}$



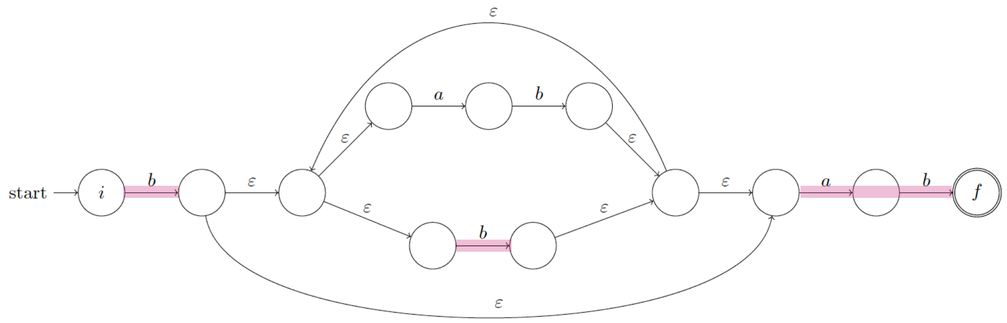
Bob holds
subset $S_2 \subseteq \{1,\ldots,m\}$
Is $S_1 \cap S_2 = \emptyset$?

- If Bob answers correctly with probability $\geq 2/3$, then Alice's message must contain $\Omega(m)$ bits
- If $S_1 = \{a_1, a_2, \ldots, a_{m'}\}$, encode as $R = \Sigma^* \{a_1 \mid a_2 \mid \ldots \mid a_{m'}\} \Sigma^*$
- If $S_2 = \{b_1, b_2, \ldots, b_{m'}\}$, encode as $T = b_1 b_2 \ldots b_{m'}$
- $S_1 \cap S_2 \neq \emptyset$ iff $T \in L(R)$
- Hence, a streaming algorithm must use $\Omega(m)$ bits, where m is the size of regexp

Dudek, Gawrychowski, Gourdel, S. [SODA'22]

- Space-efficient algorithm can only be achieved through parametrisation

$$\underbrace{b}_{1}\underbrace{(ab}_{2}\,|\,\underbrace{b)}_{3}{}^{*}\underbrace{ab}_{4}$$

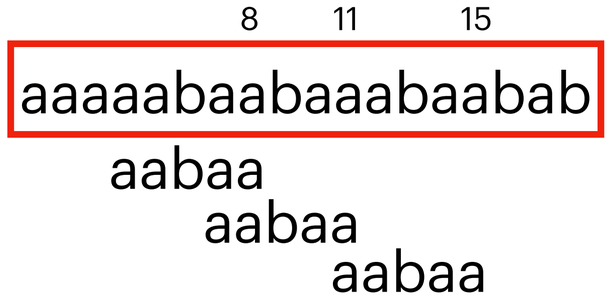


- Bille and Thorup [SODA'10]: d - number of words in the expression
- Streaming regular language membership in $O(d^3 \text{polylog } n)$ space

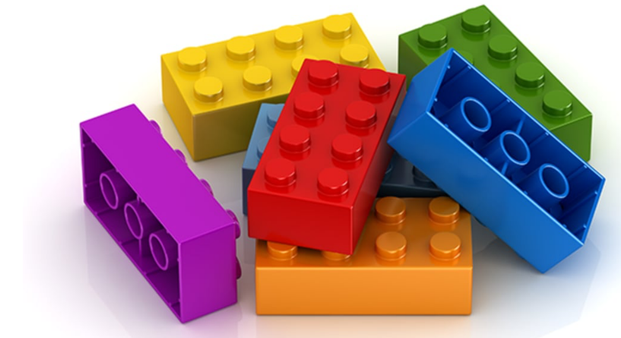
Exact pattern matching

Input: a word P of length m, a word S of length n

Output: all positions i such that a prefix $S[1..i]$ ends with P



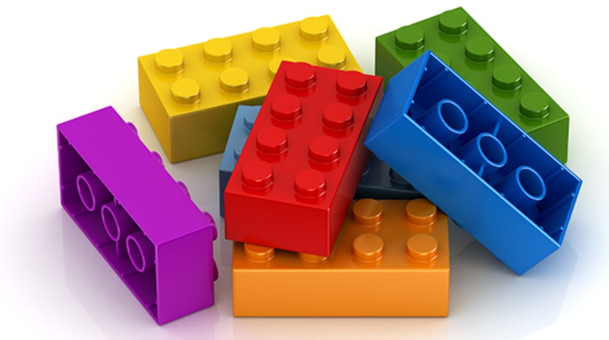
Example: P = aabaa, S = aaaaabaabaaabaabab



Streaming exact pattern matching

Porat and Porat [FOCS'09]: $O(\log m)$ space, $O(\log m)$ time algorithm

Reminder: this is <u>all space</u> used!



Approximate pattern matching

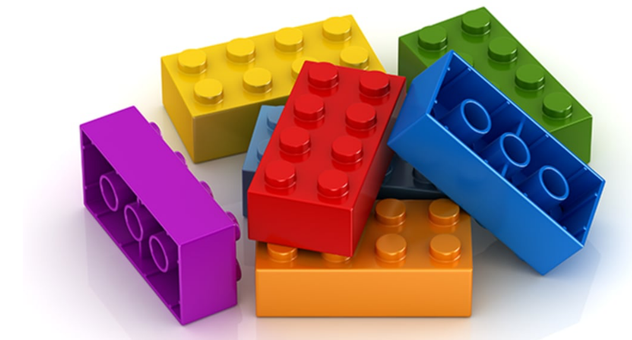
Input: a word P of length m, a word S of length n, an integer k

Output: all positions i such that a prefix $S[1..i]$ ends with an m-length subword within Hamming distance $\leq k$ from P

aaaaabaabaaabaabab

aabaa
aabaa
aabaa
aabaa
aabaa
aabaa
...

Example: P = aabaa, S = aaaaabaabaaabaabab, k = 2



Two special cases of regular expressions

- Regular expression $\Sigma^* \ (P_1 \,|\, P_2 \,|\, \ldots \,|\, P_d) \ \Sigma^*$

  Run d streaming algorithms for exact pattern matching for $P_1, \ldots, P_d$

  Space $O(d \log m)$, where $m = \max |P_i|$

  Time $O(d \log m)$

- Regular expression $\Sigma^* \ P_1 \ (a_1 \,|\, a_2 \,|\, \ldots \,|\, a_\sigma) \ P_2 \ (a_1 \,|\, a_2 \,|\, \ldots \,|\, a_\sigma) \ \ \ldots \ \ P_d \ \Sigma^*$

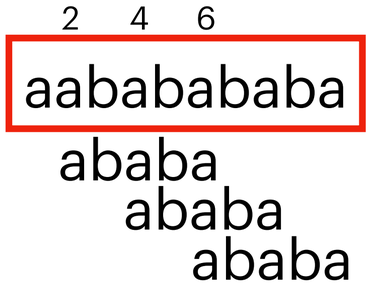
  $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$

  Run d-mismatch pattern matching algorithm for $P_1\$P_2\$ \ \ldots \ \ \$P_d$

  Space $O(d \cdot \text{polylog } m)$, where $m = \max |P_i|$

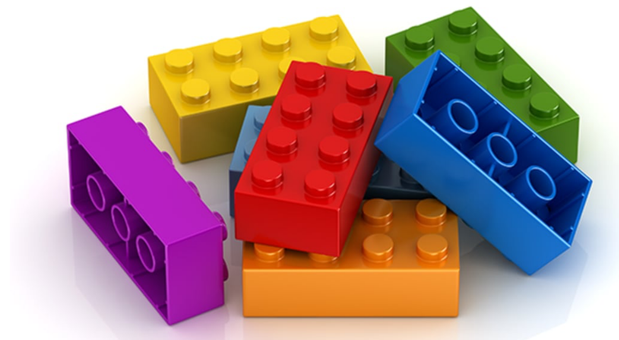
  Time $O(\sqrt{d} \cdot \text{polylog } m)$

Corollary of Fine-Wilf periodicity lemma

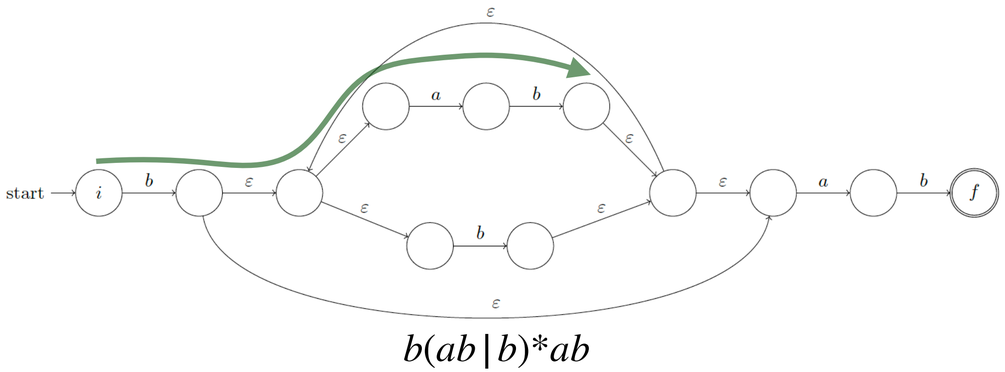


Given two strings X, Y such that $|Y| \leq 2|X|$, two cases are possible:

- Either the number of occurrences of X in Y is ≤ 2
- Or they form an arithmetic progression (X is **periodic**)



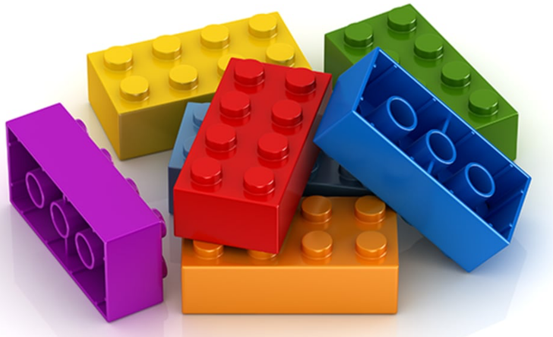
Witness



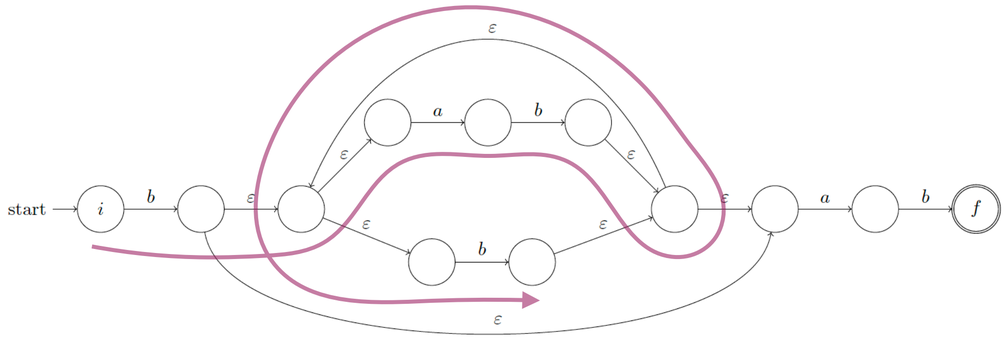
$$S = bababbaaa$$
1 2 3 4 5 6 7 8 9

- 3 is a witness for $P = ab$

- A position r is a **witness** for a word P if $S[1..r]$ is a partial occurrence of the regular expression ending with P
- If n is a witness for a word leading to a final state, S matches the regular expression



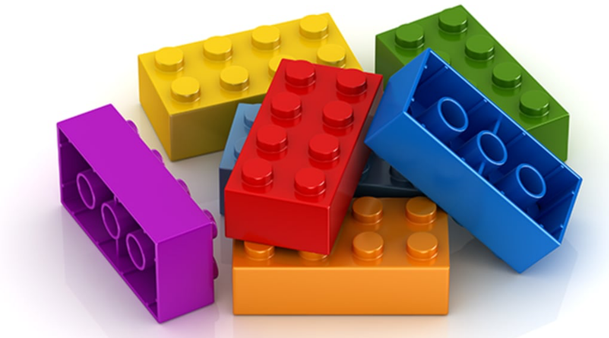
Witness



$$S = bababbaaa$$
1 2 3 4 5 6 7 8 9

- 6 is a witness for $P = b$

- A position r is a **witness** for a word P if $S[1..r]$ is a partial occurrence of the regular expression ending with P
- If n is a witness for a word leading to a final state, S matches the regular expression



High-level idea of Dudek et al. [SODA'22]

- Run the streaming pattern matching algorithm for every **power-of-two-length prefix of a word** in the regular expression (In total: $O(d \log m)$ instances)
- If $S[1..r]$ ends with an occurrence of some prefix P:
  - Decide whether r is a witness
  - Memorise it

Simplification: all prefixes are aperiodic

High-level idea of Dudek et al. [SODA'22]

- If $S[1..r]$ ends with some prefix P, $|P| = 1$:
  - Decide whether r is a witness: **easy**, enough to store all prefixes for which $T[1..r-1]$ is a witness ($O(d \log n)$ space)
  - Memorise it

Simplification: all prefixes are aperiodic

High-level idea of Dudek et al. [SODA'22]

- If $S[1..r]$ ends with some prefix P, $|P| = 2^i$:
  - Decide whether r is a witness: enough to know whether $S[1..r-2^{i-1}]$ is a witness for $P[1..2^{i-1}]$
  - For that, we can store all witnesses for $P[1..2^{i-1}]$ among the 2^i latest positions ($O(1)$ as it is **aperiodic**)

Simplification: all prefixes are aperiodic

High-level idea of Dudek et al. [SODA'22]

- To get rid of the simplification, we use the fact that for periodic prefixes their occurrences form an arithmetic progression that can be stored in $O(1)$ space
- Unfortunately, things are **not that easy**: not all occurrences in the progression are witnesses
- We store witnesses in the **beginning of a progression** only
- And then jump from the stored witnesses to the current position using a graph-based mechanism

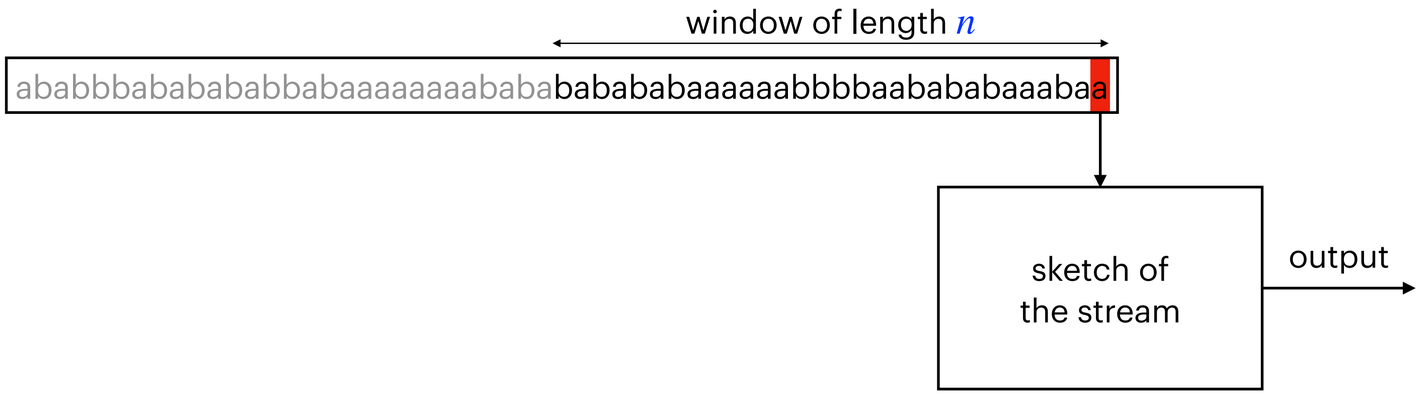
For a **number of words** d in a regular expression:

- $\mathrm{poly}(d, \log n)$-space streaming algorithm (Dudek et al., SODA'22)

Sliding window

based on Ganardi et al. “Regular Languages in the Sliding Window Model” TheoretiCS 4 (2025)
and (partially) on slides by Ganardi

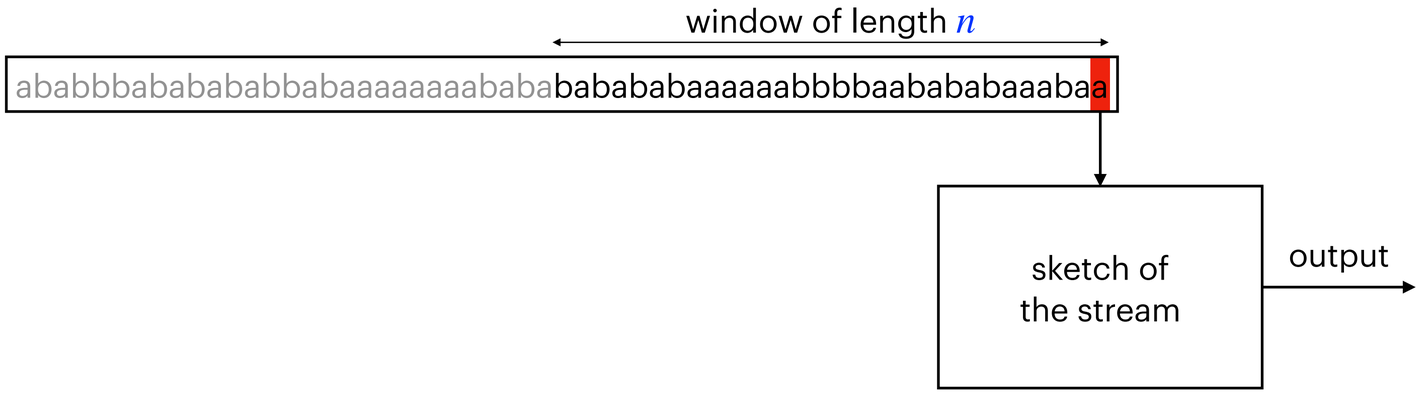
Regular language recognition



When a new letter arrives:

- ACCEPT if the n-length suffix (sliding window) is in the language
- REJECT otherwise

Regular language recognition



- $R_1 = \{a|b|c\}^*$

  $O(1)$ bits of space
- $R_2 = \{a|b|c\}^* \; a \; \{a|b|c\}^*$

  $O(\log n)$ bits of space (store the position of the last a up to threshold n)
- $R_3 = a \; \{a|b|c\}^*$

  $O(n)$ bits of space

Lower bound for R_3

- Lower bound: If the answer of Bob is correct with probability $\geq 2/3$, Alice's message contains $\Omega(n)$ bits

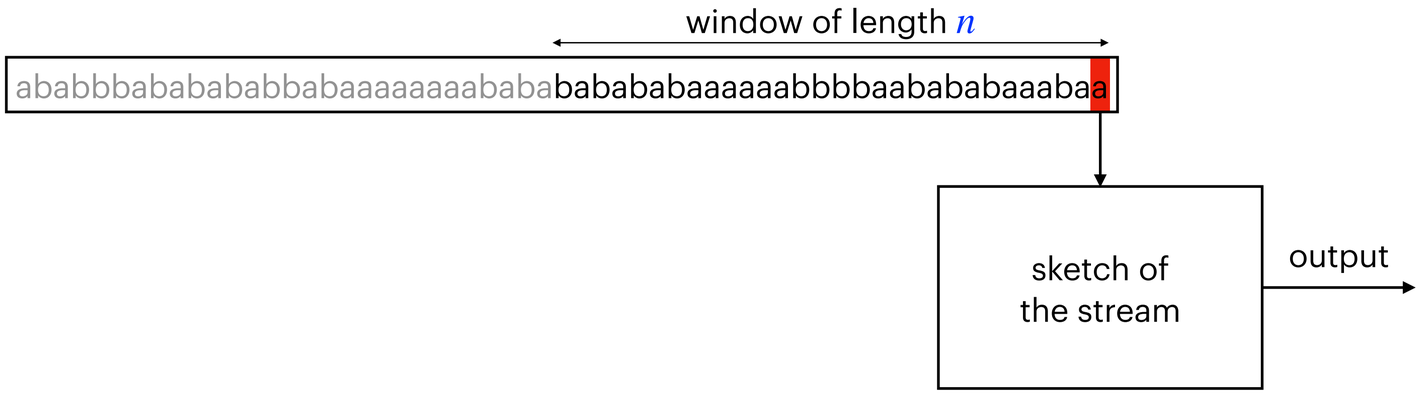
Lower bound for R_3

If there is a streaming algorithm for $R_3 = a\ \{a \mid b \mid c\}^*$, we can solve Index:

- Alice runs the algorithm on her string, sends the memory of the algorithm to Bob
- Bob continues running the algorithm on a^{i-1}
- If the latest n-length suffix matches R_3, the i-th letter is a

The algorithm must use $\Omega(n)$ bits of space

Regular language recognition



From now on, assume a constant-size regular expression

Space (bits)	Deterministic streaming
$O(1)$	<Suffix testable, Length>
$O(\log n)$	<Left ideals, Length>
$O(n)$	Reg

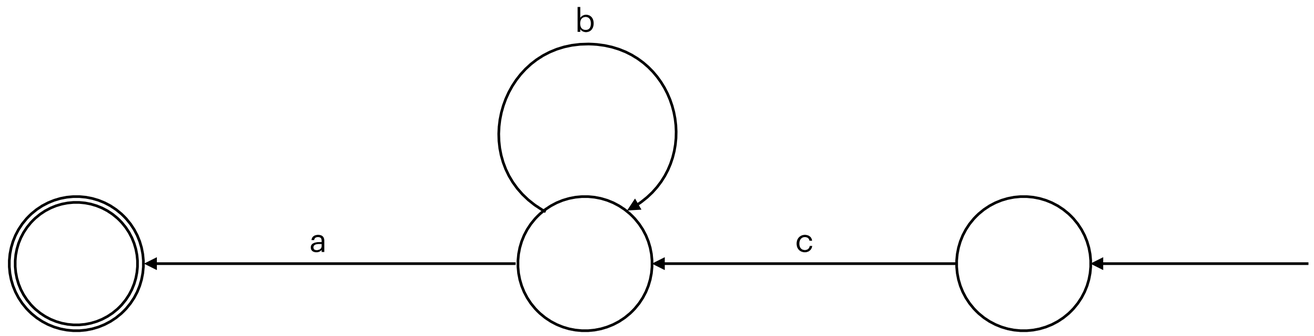
Suffix testable:
finite Boolean combinations of $\Sigma^* w$, w - a word

Length:
$\forall n$, either all n-length words are in L, or not

Left ideals:
$\Sigma^* L'$, where L' - regular

Right-deterministic finite automata (right-DFA)

= deterministic finite automata for the reversal language L^R

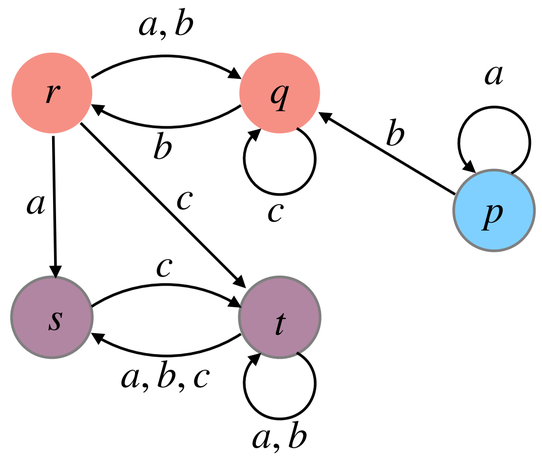


Example: $L = L(ab^*c)$

Path summary of a run

Path summary of a run

For every visited SCC, store the entry state and the subrun length



$$t \xrightarrow{b} s \xrightarrow{a} r \xrightarrow{b} q \xrightarrow{c} q \xrightarrow{c} q \xrightarrow{b} p \xrightarrow{a} p \xrightarrow{a} p$$

$$(1,s)(4,q)(3,p)$$

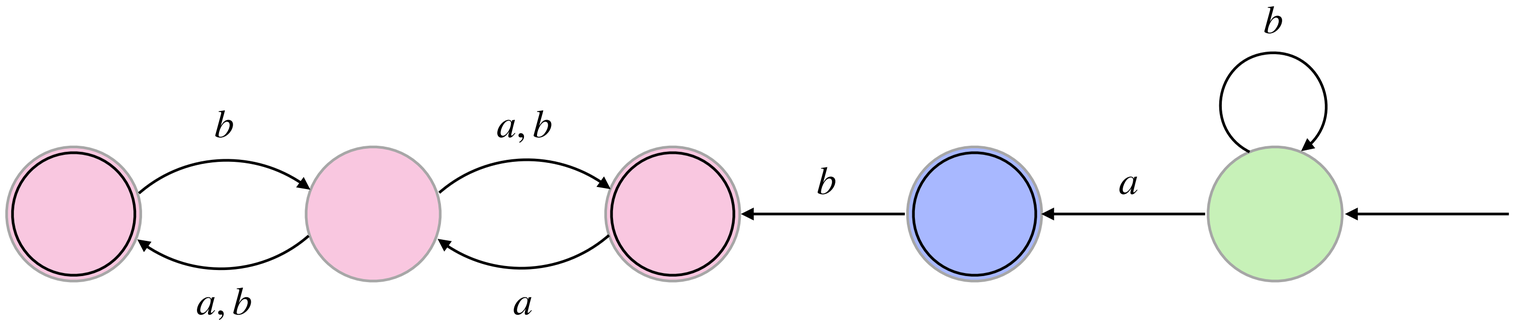
Maintain the path summaries of all $|Q|$ runs on the sliding window in $O(|Q|^2 \log n) = O(\log n)$ bits.

Well-behaved sets

A subset $C \subseteq Q$ is well-behaved if

for all $p, q, r \in C$ and all C-runs $q \overset{u}{\leftarrow} p$ and $r \overset{v}{\leftarrow} p$ of equal length we have:

q is final $\Leftrightarrow$ r is final



Logarithmic vs. linear

Theorem

1. If Q is well-behaved, there is a $O(\log n)$-space streaming algorithm;
2. Otherwise, there are infinitely many n for which $\Omega(n)$ bits are needed.

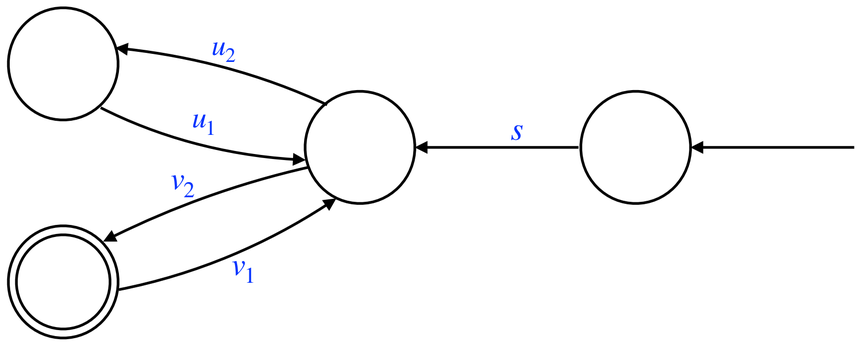
1. The path summary of the run on the sliding window from the initial state determines whether the sliding window is accepted - $O(\log n)$ bits

Logarithmic vs. linear

Theorem

1. If Q is well-behaved, there is a $O(\log n)$-space streaming algorithm;
2. Otherwise, there are infinitely many n for which $\Omega(n)$ bits are needed.

2. The automaton contains the following pattern where $|u_1| = |u_2|$ and $|v_1| = |v_2|$:



Setting $u = u_1 u_2$ and $v = v_1 v_2$ yields

$u_2\{u, v\}^* s \cap L = \varnothing$ and $v_2\{u, v\}^* s \subset L$

$\Rightarrow \Omega(n)$ space by reduction from **Index**

Constant vs. logarithmic

$U(Q)$ = the set of states reachable by runs of unbounded length.

Theorem

1. If $U(Q)$ is well-behaved, there is a $O(1)$-space streaming algorithm;
2. Otherwise, there are infinitely many n for which $\Omega(\log n)$ bits are needed.

Language-theoretic characterisations

A subset $C \subseteq Q$ is well-behaved if for all $p, q, r \in C$ and all C-runs $q \xleftarrow{u} p$ and $r \xleftarrow{v} p$ of equal length we have:

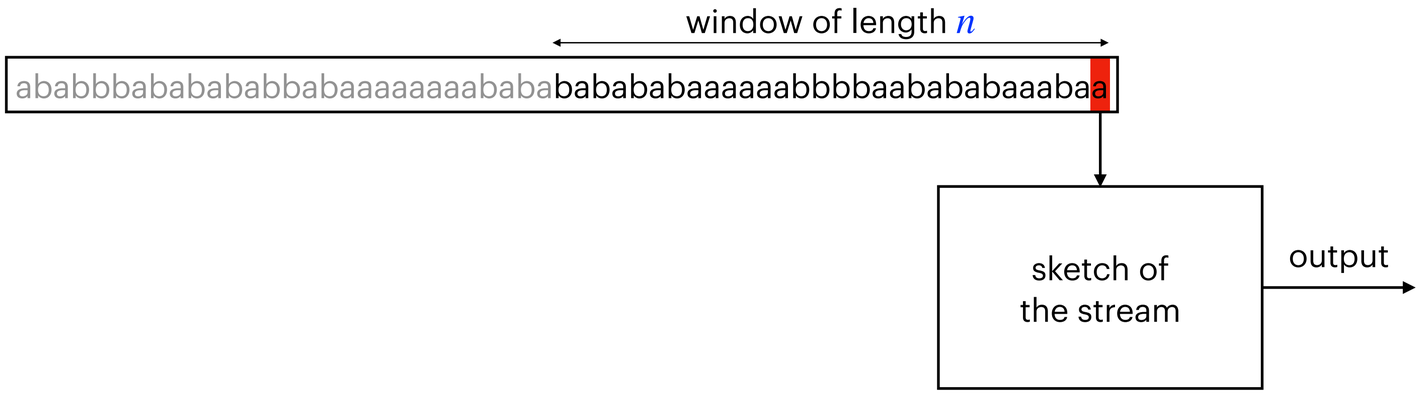
q is final $\Leftrightarrow$ r is final

$U(Q)$ = the set of states reachable by runs of unbounded length.

Theorem

1. Q is well-behaved $\Leftrightarrow$ L is a finite Boolean combination of regular left ideals $\Sigma^* L'$ and regular length languages
2. $U(Q)$ is well-behaved $\Leftrightarrow$ L is a finite Boolean combination of suffix testable languages (finite Boolean combinations of $\Sigma^* w$) and regular length languages.

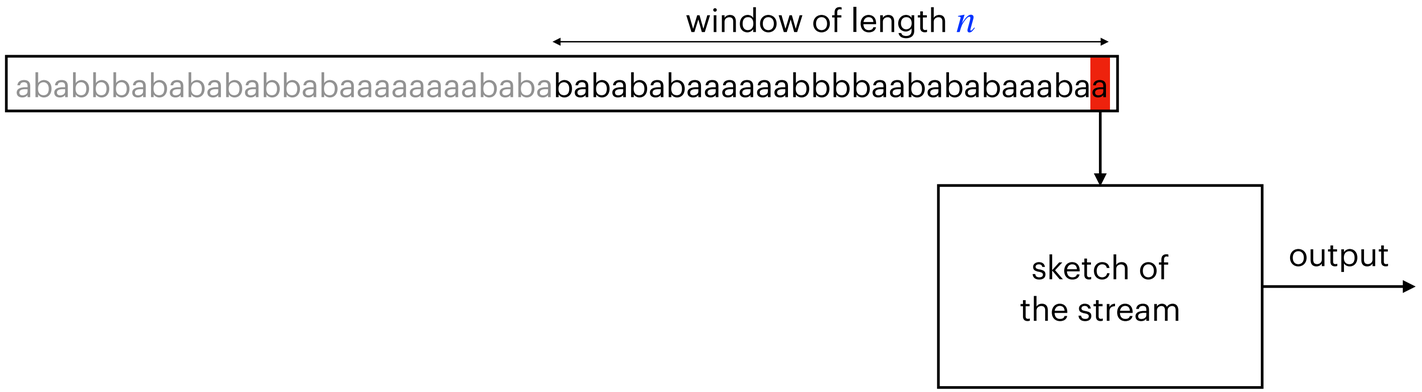
Regular language recognition



From now on, assume a constant-size regular expression

Space (bits)	Deterministic streaming	Randomised streaming
$O(1)$	<Suffix testable, Length>	<Suffix testable, Length>
$O(\log \log n)$	-	<Suffix testable, Suffix free, Length>
$O(\log n)$	<Left ideals, Length>	<Left ideals, Length>
$O(n)$	Reg	Reg

Randomised streaming algorithms



- Randomised algorithms decide whether the sliding window $\in L$ correctly with probability $\geq 2/3$ (two-sided error)
- Every regular suffix-free language has a randomised streaming algorithm using $O(\log \log n)$ bits
- L is suffix-free if $uv \in L$ and $u \neq \varepsilon$ implies $v \notin L$

Suffix-free languages

Example: $L = L(ab^*)$

Decide whether the position of the most recent a-letter is n using two probabilistic counters.

1. Modulo counter M_n

- Maintain pos modulo a random prime p with $O(\log \log n)$ bits
- Accept if pos $\equiv n \pmod{p}$
- If pos $\neq n$ and pos $< 2n$ then M_n rejects with high probability

Problem: if pos $\geq 2n$

Suffix-free languages

2. Bernoulli counter Z_n

- Initialise $x := \text{high}$
- On input a: set $x := \text{low}$
- On input b: set $x := \text{high}$ with probability $1/2n$
- Accept if $x = \text{low}$

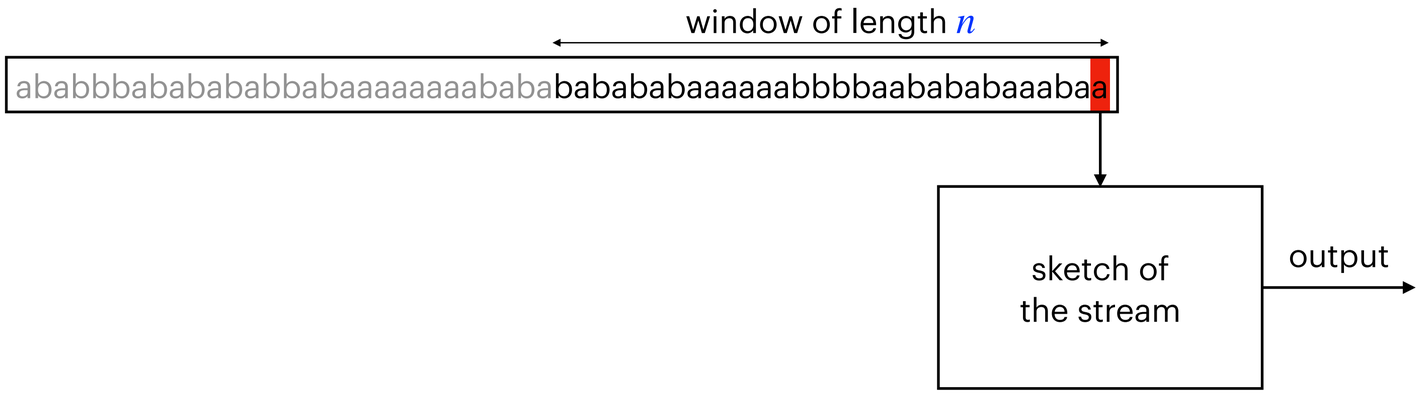
If pos $\geq 2n$ then

$$Pr[x = \text{high}] \geq 1 - (1 - \frac{1}{2n})^{2n} \to 1 - 1/e \approx 0.6321$$

If pos $\leq n$ then

$$Pr[x = \text{low}] \geq 1 - (1 - \frac{1}{2n})^{n} \to 1/\sqrt{e} \approx 0.6065$$

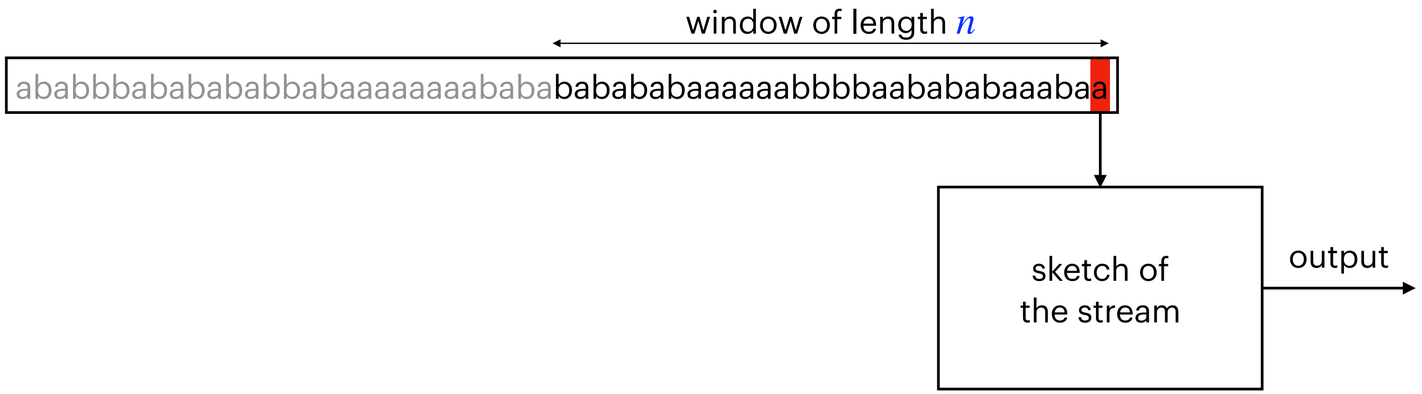
Regular language recognition



From now on, assume a constant-size regular expression

Space (bits)	Deterministic streaming	Randomised streaming
$O(1)$	<Suffix testable, Length>	<Suffix testable, Length>
$O(\log \log n)$	-	<Suffix testable, Suffix free, Length>
$O(\log n)$	<Left ideals, Length>	<Left ideals, Length>
$O(n)$	Reg	Reg

Regular language recognition



When a new letter arrives:

- ACCEPT if the n-length suffix (sliding window) is in the language
- REJECT if the sliding window is **far** the language
- ACCEPT or REJECT otherwise

Two-sided sliding window property tester

- Maintain a sample of random substrings of the current n-length suffix of the stream (Braverman et al., J. Comput. Syst. Sci.'12)
- For every letter sampled, we pay $O(\log n)$ bits of space. The sample can be smaller with probability $1/3$.
- Space = $O_\varepsilon(\log n)$ bits

- Feed in the sample into a property tester for regular languages
- If the input is in the language, the algorithm ACCEPTS with constant probability.
  If at least ε-fraction of the letters must be modified to obtain a string in the language, the algorithm REJECTS with constant probability.

Sliding window property testers

Corollary

There is a two-sided sliding window property tester which uses $O_\varepsilon(\log n)$ bits of space

What about deterministic testers? one-sided? two-sided?

Sliding window property testers

Deterministic

- Constant Hamming gap ("far" = need to modify a constant number of letters)
- $O(\log n)$ bits of space

One-sided

- Hamming distance gap = εn
- $O_\varepsilon(1)$, $O_\varepsilon(\log \log n)$, or $O_\varepsilon(\log n)$ bits

Two-sided

- Hamming distance gap = εn
- $O_\varepsilon(1)$ bits

Deterministic streaming property tester

- For all $q \in Q$, maintain the path summary of the run on the sliding window starting at q
- $Acc(q)$ = all n such that for some n-length word, the run on it starting at q ends in a final state
- If the path summary for the initial state is

$$(\ell_m, q_m), (\ell_{m-1}, q_{m-1}), \ldots, (\ell_1, q_1)$$

and $\ell_m \in Acc(q_m)$, accept, else, reject.

Deterministic streaming property tester

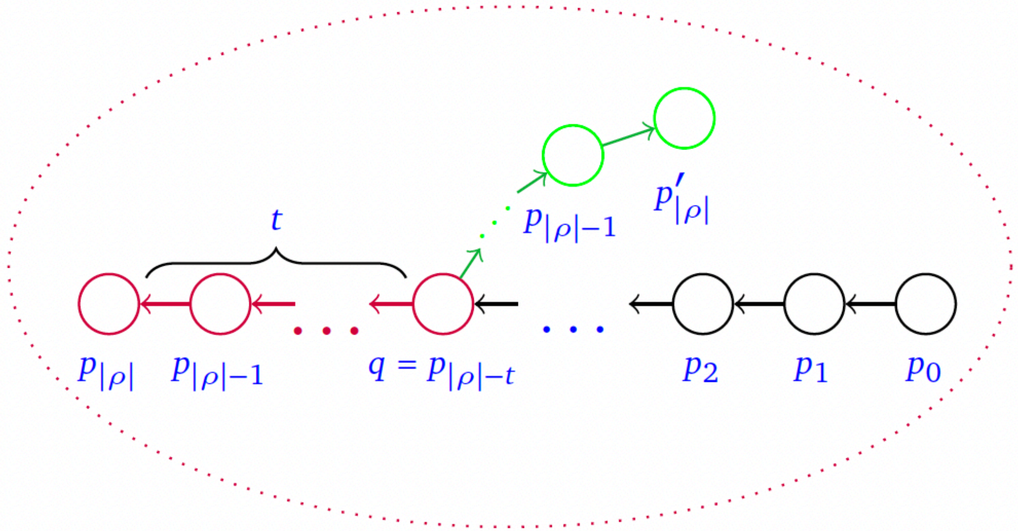
Alon et al., SIAM J. of Comput.'00

Consider an SCC C, and let d be the greatest common divisor of all cycle lengths. There exists a partition $C(V) = \cup_{i=0}^{d-1} V_i$ such that:

For all $u \in V_i$ and $v \in V_j$:

- length of any path from u to v is $(j - i) \mod d := \text{shift}(u, v)$
- for all large enough $r = \text{shift}(u, v) \mod d$, there is a run from u to v of length r

Deterministic streaming property tester

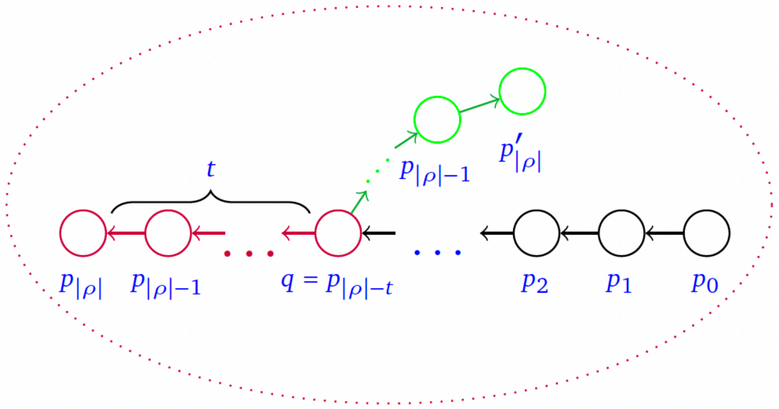


$X =_t Y$ if for all $z \geq t$: $z \in X \Leftrightarrow z \in Y$

There is t such that for all non-transient SCC C and all $p, q \in C$:

- $Acc(q) =_t Acc(q) + d$ ("periodicity" ⇒ can be stored in $O(1)$ space)
- $Acc(p) =_t Acc(q) + \text{shift}(p, q)$

Deterministic streaming property tester



If ρ is a run in SCC, $t \leq |\rho| \in Acc(p_0)$, labeled by a word w, then one can change at most first t letters of w so that the resulting word labels an accepting run.

$$Acc(q) =_t Acc(p_0) + \mathsf{shift}(q, p_0)$$

$$\mathsf{shift}(q, p_0) = -\mathsf{shift}(p_0, q) = -(|\rho| - t) \mod d$$

$t = |\rho| - (|\rho| - t) \in Acc(q)$, hence there is an accepting run from q of length t

Deterministic streaming property tester

- For all $q \in Q$, maintain the path summary of the run on the sliding window starting at q [$O(\log n)$ space]
- $Acc(q)$ = all n such that for some n-length word, the run on it starting at q ends in a final state
- If the path summary for the initial state is

$$(\ell_m, q_m), (\ell_{m-1}, q_{m-1}), \ldots, (\ell_1, q_1)$$

and $\ell_m \in Acc(q_m)$, accept, else, reject.

If the sliding window is in the language, $\ell_m \in Acc(q_m)$.

If the tester accepts, then $\ell_m \in Acc(q_m)$ and one can apply re-rerouting to change at most t first letters of the sliding window to obtain a word in the language.

Thank you!