Algorithms for string processing in restricted-access models of computation

Tatiana Starikovskaya

Modern string processing

Applications

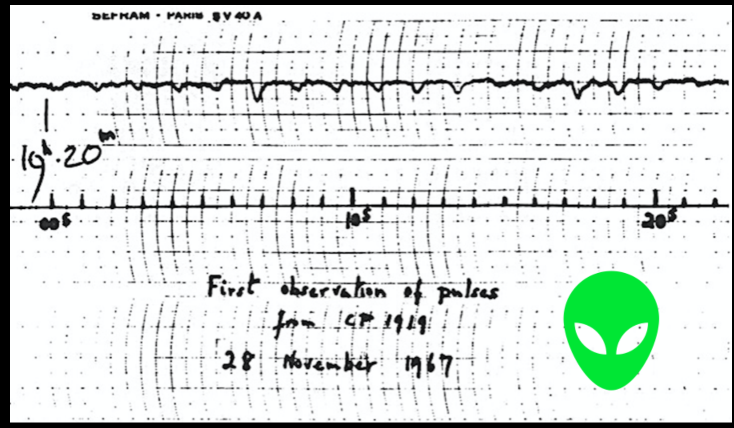
- Bioinformatics
- Information Retrieval
- Cyber Security
- ...

Classical approaches

- We can afford to store the input data in full
- We can read the whole input

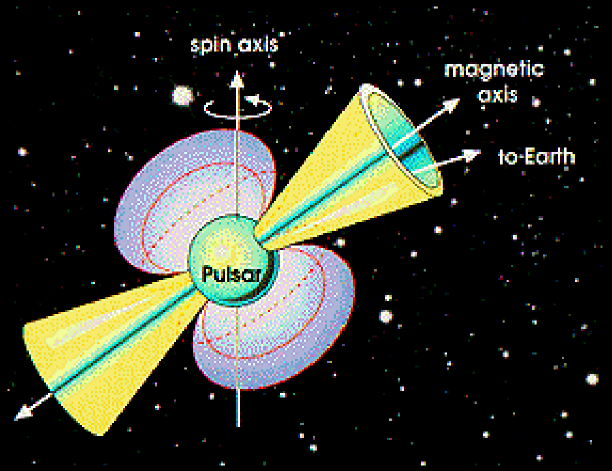
In the Big Data world, we must do better.

Little Green Men or pulsars



In 1968, Jocelyn Bell Burnell and her supervisor, Anthony Hewish, detected a strange periodic signal. They nicknamed their discovery LGM-1 for **Little Green Men**.

Little Green Men or pulsars



In fact, it was the first experimental observation of a pulsar — a rotating star that emits beams of electromagnetic radiation out of its magnetic poles.

Little Green Men or pulsars

- Pulsars suggest a new way of keeping track of time by observing a collection of pulsars
- Pulsars allow to study the very early universe and colliding supermassive black holes at the centres of galaxies
- Double pulsar systems allow to study Einstein's general theory of relativity

Question: Given a stream of integers, find its period.

Validity of XML documents

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="first.css"?>
<!DOCTYPE DOCUMENT [
  <!ELEMENT DOCUMENT (GREETING, MESSAGE)>
  <!ELEMENT GREETING (#PCDATA)>
  <!ELEMENT MESSAGE (#PCDATA)>
]>
<DOCUMENT>
  <GREETING>
    Hello From XML
  </GREETING>
  <MESSAGE>
    Welcome to the wild and woolly world of XML.
  </MESSAGE>
</DOCUMENT>
```

Question: Given a (long) string, can we (quickly) decide if it is a valid XML document?

Fire alarm

Question: Given an integer n, a stream of characters and a regular language $L = \Sigma^* \circ [\{H\ \Sigma^*\ S\} \cup \{S\ \Sigma^*\ H\}] \circ \Sigma^*$. When a new character arrives, report YES if the latest n-length suffix of the stream belongs to L.

Restricted data access models

Streaming algorithms

We receive the input as a stream, and must process it on-the-fly.
We do not want to store the data in full (requires too much space).

Restricted data access models

Streaming algorithms

We receive the input as a stream, and must process it on-the-fly. We do not want to store the data in full (requires too much space).

Property testing algorithms

We must decide if the input has a property $\mathcal{P}$. We can only read a small part of the input (reading the whole input requires too much time).

A relaxation: return YES if the input satisfies $\mathcal{P}$, and NO if the distance from the input to any string satisfying $\mathcal{P}$ is large. Otherwise, YES / NO.

Restricted data access models

Streaming algorithms

We receive the input as a stream, and must process it on-the-fly. We do not want to store the data in full (requires too much space).

Property testing algorithms

We must decide if the input has a property $\mathcal{P}$. We can only read a small part of the input (reading the whole input requires too much time).

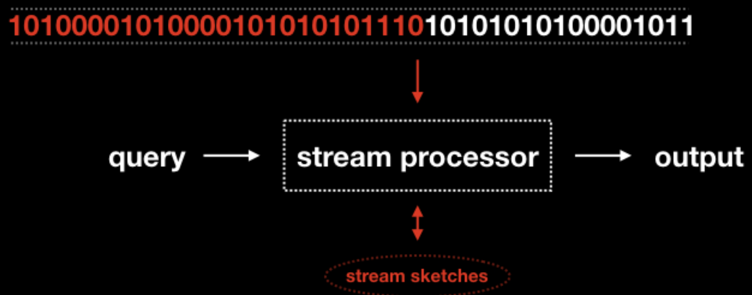
A relaxation: return YES if the input satisfies $\mathcal{P}$, and NO if the distance from the input to any string satisfying $\mathcal{P}$ is large. Otherwise, YES / NO.

Streaming property testing algorithms

We must decide whether a stream or its part has a property $\mathcal{P}$. We cannot store the data in full, but can relax the requirements as above.

Part I: Streaming algorithms

Streaming algorithms



Objectives: real time & small space
When to use: stream data, big data

Streaming algorithms for string processing

Pattern matching

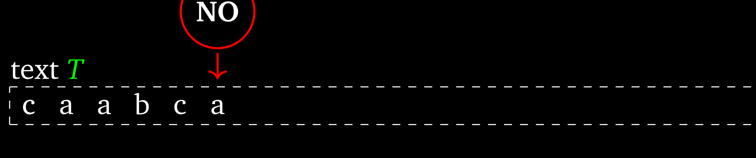
- exact [PP09, BG14, CFP+15, GP17]
- approximate [PP09, CFP+16, CS16, Sta17, GS19]

Periodicity [EJS10, EGAZ17, EGAZ18, GRS19]

Palindromes, runs, repeats [GMSU19, MS19a, MS19b]

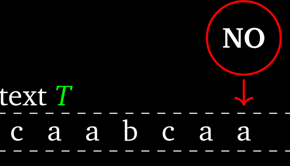
Language recognition [BLRV13, MMN14, GHL16, GHL18a, GHL18b, GHK+18, GJL18, Gan19]

Exact pattern matching



- **Query** = "Is there an occurrence of P?"
- **Space** = total space used by the stream processor

Exact pattern matching



b c a a a c

pattern P

- **Query** = "Is there an occurrence of P?"
- **Space** = total space used by the stream processor

Exact pattern matching

NO

text T

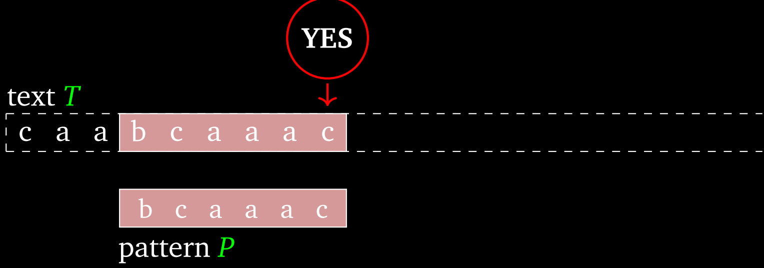
c a a b c a a a

b c a a a c

pattern P

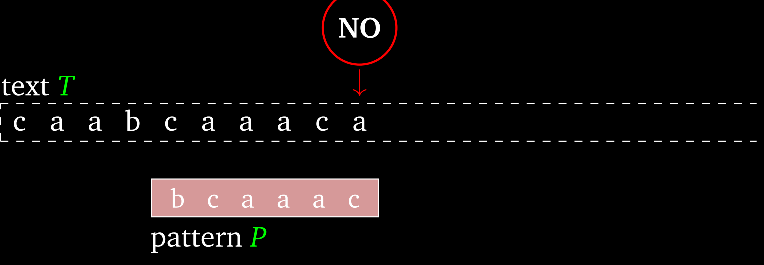
- **Query** = "Is there an occurrence of P?"
- **Space** = total space used by the stream processor

Exact pattern matching



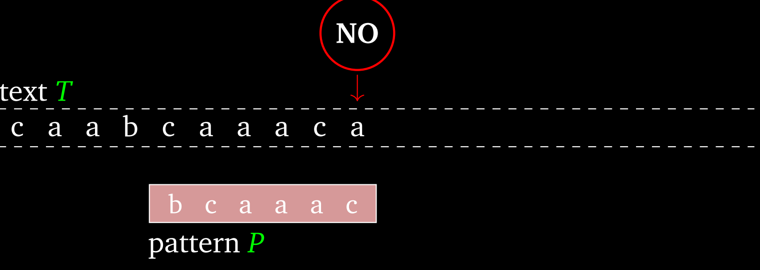
- **Query** = "Is there an occurrence of P?"
- **Space** = total space used by the stream processor

Exact pattern matching



- **Query** = "Is there an occurrence of P?"
- **Space** = total space used by the stream processor

Exact pattern matching



- **Query** = "Is there an occurrence of P?"
- **Space** = total space used by the stream processor

Theorem. Given a streaming text and a pattern of length m. There is an algorithm that solves the exact pattern matching problem in $\mathcal{O}(\log m)$ space and $\mathcal{O}(\log m)$ time per arriving character. The algorithm is correct with high probability.

Reminder: Karp–Rabin algorithm

Karp–Rabin fingerprint

$$\varphi(s_1 s_2 \ldots s_m) = \sum_{i=1}^{m} s_i \cdot r^{m-i} \bmod p$$

where p is a prime and r is a random integer $\in [0, p-1]$

It's a good hash function:

S_1, S_2 are two strings of length m, the prime p is large

- If $S_1 = S_2$, then $\varphi(S_1) = \varphi(S_2)$
- If $S_1 \neq S_2$, then $\varphi(S_1) \neq \varphi(S_2)$ w.h.p.

Karp–Rabin algorithm

YES

text T

c a a b c a a a c a

b c a a a c

pattern P

When a new character $t_i = a$ arrives:

1. Update $\varphi(t_{i-m+1} \dots t_{i-1} t_i) = \sum_{i=1}^{m} t_{i-m+j} \cdot r^{m-j} \bmod p$:

$$\varphi(t_{i-m+1} \dots t_{i-1} t_i) = (\varphi(t_{i-m} \dots t_{i-1} t_{i-1}) - t_{i-m} \cdot r^{m-1}) \cdot r + t_i \bmod p$$

2. If $\varphi(t_{i-m+1} \dots t_{i-1} t_i) = \varphi(P)$, output “YES”

We need t_{i-m} to update the fingerprint $\Rightarrow$ **we must store** $t_{i-m}, \dots, t_{i-1}$

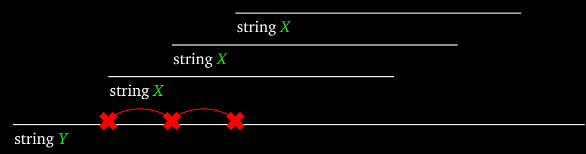
Streaming exact pattern matching

The period

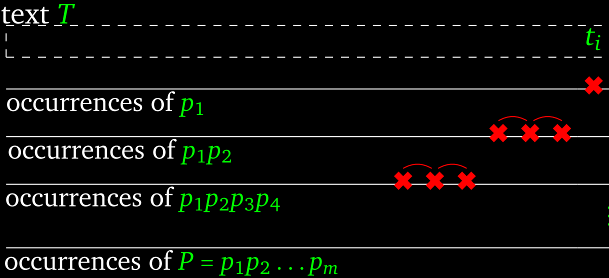
The period of a string X is the smallest integer $p \geq 1$ such that $X[i] = X[i+p]$ for all i.

Progression of occurrences

If $|Y| = 2|X|$, and Y contains ≥ 3 occurrences of X, they form an arithmetic progression with difference equal to the period of X.



Streaming exact pattern matching



In level j, we store occurrences of $p_1p_2 \dots p_{2^j}$ in $T[i - 2^{j+1} + 1, i]$. They form an arithmetic progression. We store:

- Number of occurrences
- The leftmost and the second leftmost positions lp, lp'
- The fingerprints $\varphi(t_1t_2 \dots t_{lp-1})$, $\varphi(t_{lp} \dots t_{lp'-1})$, $\varphi(t_1 \dots t_i)$

Streaming exact pattern matching

text T

occurrences of p_1

occurrences of p_1p_2

occurrences of $p_1p_2p_3p_4$

$\vdots$

occurrences of $P = p_1p_2 \dots p_m$

for each character t_i **do**
  if $t_i = p_1$ **then** push i to level 0
  for each $j = 0, \dots, \log m - 1$
    $lp \leftarrow$ leftmost position in level j
    if $i - lp + 1 = 2^{j+1}$ **then**
      Pop lp from level j
      if $\varphi(t_{lp} \dots t_i) = \varphi(p_1 \dots p_{2^{j+1}})$ **then** push lp to level $j+1$

Streaming exact pattern matching

text T

t_i

occurrences of p_1

occurrences of $p_1 p_2$

occurrences of $p_1 p_2 p_3 p_4$

⋮

occurrences of $P = p_1 p_2 \dots p_m$

for each character t_i **do**
  if $t_i = p_1$ **then** push i to level 0
  for each $j = 0, \dots, \log m - 1$
    $lp \leftarrow$ leftmost position in level j
    if $i - lp + 1 = 2^{j+1}$ **then**
      Pop lp from level j
      if $\varphi(t_{lp} \dots t_i) = \varphi(p_1 \dots p_{2^{j+1}})$ **then** push lp to level $j + 1$

Streaming exact pattern matching

text T

t_i

If i is an occ. of p_1, push it to level 0

occurrences of p_1

occurrences of p_1p_2

occurrences of $p_1p_2p_3p_4$

$\vdots$

occurrences of $P = p_1p_2 \ldots p_m$

for each character t_i **do**
  if $t_i = p_1$ **then** push i to level 0
  for each $j = 0, \ldots, \log m - 1$
    $lp \leftarrow$ leftmost position in level j
    if $i - lp + 1 = 2^{j+1}$ **then**
      Pop lp from level j
      if $\varphi(t_{lp} \ldots t_i) = \varphi(p_1 \ldots p_{2^{j+1}})$ **then** push lp to level $j + 1$

Streaming exact pattern matching

text T

t_i

occurrences of p_1

occurrences of $p_1 p_2$

occurrences of $p_1 p_2 p_3 p_4$

⋮

occurrences of $P = p_1 p_2 \dots p_m$

If i is an occ. of p_1, push it to level 0

for each character t_i **do**
  if $t_i = p_1$ **then** push i to level 0
  for each $j = 0, \dots, \log m - 1$
    $lp \leftarrow$ leftmost position in level j
    if $i - lp + 1 = 2^{j+1}$ **then**
      Pop lp from level j
      if $\varphi(t_{lp} \dots t_i) = \varphi(p_1 \dots p_{2^{j+1}})$ **then** push lp to level $j + 1$

Streaming exact pattern matching

text T

t_i

occurrences of p_1

occurrences of $p_1 p_2$

occurrences of $p_1 p_2 p_3 p_4$

⋮

occurrences of $P = p_1 p_2 \dots p_m$

If lp is an occ. of $p_1 p_2$, promote it

for each character t_i **do**
  if $t_i = p_1$ **then** push i to level 0
  for each $j = 0, \dots, \log m - 1$
    $lp \leftarrow$ leftmost position in level j
    if $i - lp + 1 = 2^{j+1}$ **then**
      Pop lp from level j
      if $\varphi(t_{lp} \dots t_i) = \varphi(p_1 \dots p_{2^{j+1}})$ **then** push lp to level $j + 1$

Streaming exact pattern matching

text T

t_i

occurrences of p_1

occurrences of $p_1 p_2$

occurrences of $p_1 p_2 p_3 p_4$

⋮

occurrences of $P = p_1 p_2 \dots p_m$

If lp is an occ. of $p_1 p_2$, promote it

for each character t_i **do**
  if $t_i = p_1$ **then** push i to level 0
  for each $j = 0, \dots, \log m - 1$
    $lp \leftarrow$ leftmost position in level j
    if $i - lp + 1 = 2^{j+1}$ **then**
      Pop lp from level j
      if $\varphi(t_{lp} \dots t_i) = \varphi(p_1 \dots p_{2^{j+1}})$ **then** push lp to level $j + 1$

Streaming exact pattern matching

text T ... t_i

occurrences of p_1

occurrences of p_1p_2

occurrences of $p_1p_2p_3p_4$

⋮

occurrences of $P = p_1p_2 \ldots p_m$

For each level we need:

- $O(1)$ space
- $O(1)$ time for updating and extracting $\varphi(t_{lp} \ldots t_i)$

In total, the algorithm uses $O(\log m)$ space and $O(\log m)$ time per character.

Periodicity in streams

Given a stream of characters, compute its period

Motivation:

- Detecting pulsars!
- Understanding the structure of streams
- Detecting anomalies in streams

Exact periods [EJS10]

- Periodic streams: $O(\log n)$ space, $O(\log n)$ time
- Non-periodic streams: $\Omega(n)$ space

Approximate periods (Hamming distance $\leq k$) [EGAZ17]

- Periods of length $< n/2$: $\tilde{O}(k^4)$ space
- All periods: $\Omega(n)$ space

Approximate periods ($\leq k$ wildcards) [EGAZ18]

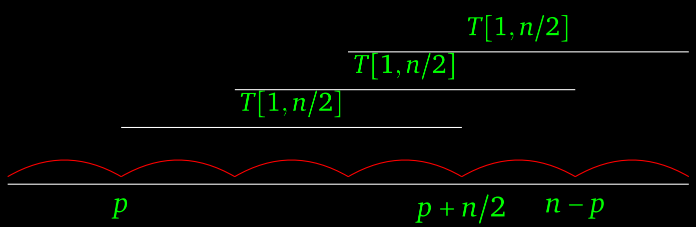
- Periods of length $< n/2$: $\tilde{O}(k^3)$ space
- All periods: $\Omega(n)$ space

Quasiperiods [GRS19]

- Periods can overlap

Exact periods

We will show how to compute the period if it is $\leq n/4$

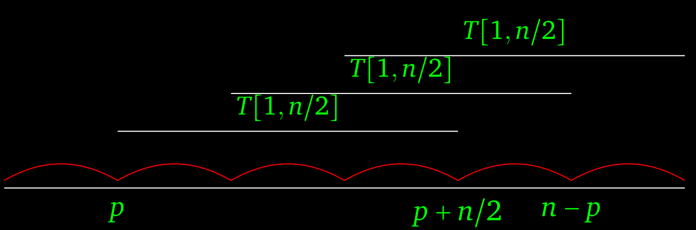


Lemma The period is equal to p iff $T[1,n-p] = T[p+1,n]$

Lemma The only candidate for the period p is the first occurrence of $T[1,n/2]$ in T

Exact periods

We will show how to compute the period if it is $\leq n/4$



Algorithm:

- Use the exact pattern matching algorithm to find the first occurrence p of $T[1,n/2]$, as well as $\varphi(T[1,p])$ (happens when $T[p+n/2-1]$ arrives)
- Memorize $\varphi(T[1,n-p])$ (we have $n-p \geq p+n/2$)
- If $\varphi(T[1,n-p]) = \varphi(T[p+1,n])$, then p is the period w.h.p

Part II: Property testing

Property testers

Task: Decide whether the input T has a property $\mathcal{P}$

Property tester algorithm must

- ACCEPT, if T satisfies $\mathcal{P}$
- REJECT, if T if ε-far from satisfying $\mathcal{P}$
- ACCEPT or REJECT otherwise

ε-far = we must fix $\geq \varepsilon n$ characters of T so that $\mathcal{P}$ is satisfied

Time: the number of queried characters

When to use property testing?

- It is infeasible to fully recover the input
- Inputs that are close to having the property are good enough
- Testing as a preliminary step before deciding
- Testing as a preliminary step before reconstructing
- In the third part we will see that property testers can be used to build space-efficient streaming algorithms

Property testing in string processing

Pattern matching [BEKR17]

Membership in regular languages [AKNS00, NLN13, NLN15]

Well-parenthesised sequences [AKNS00, PRR01, FMS18]

Property testing for well-parenthesised sequences

D_m — balanced sequences of parentheses of m types

✔ ()([]())[]([])   ✗ ()(([][)()((([]

	Time	**Reference**
D_1	Const.	[AKNS00]
$D_m, m \geq 2$	$cn^{1/11} \leq T \leq Cn^{2/3}$	[PRR01]
	$cn^{1/5} \leq T \leq Cn^{2/5+\delta}$	[FMS18]

Why is it interesting?

D_m — well-balanced sequences of parentheses of m types

Simplicity: simplest context-free language

Practicality: processing of semi-structured documents

Universality: any context-free language can be expressed through it (Chomsky-Schützenberger theorem)

Tester of Parnas et al. [PRR01]

D_m — well-balanced sequences of parentheses of m types

$$()(\{()\})([])\{((([]())([])([\{\}]())([])))\}$$

Tester of Parnas et al. [PRR01]

D_m — well-balanced sequences of parentheses of m types

✔ $()(\{()\})([])\{((([]())([])([\{\}]())([])))\}$

Tester of Parnas et al. [PRR01]

D_m — well-balanced sequences of parentheses of m types

✔ ()({()})([]){((([]())([])([{}]())([])))}

Does not look like a simple property to test!

Main idea: reduce to testing several simple properties

Tester of Parnas et al. [PRR01]

D_m — well-balanced sequences of parentheses of m types

✔ ()({()})([]){((([]())([])([{}]())([])))}

☛ If we replace all opening parentheses with “(”, and all closing parentheses with “)”, the resulting string must be in D_1

✔ ()((()))(())((((()())(())((())())(()))))

And we know how to test this property in $\mathcal{O}(1)$ time [AKNS00]

Not sufficient: ✗ ()({{)}) becomes ✔ ()((()))

Tester of Parnas et al. [PRR01]

D_m — well-balanced sequences of parentheses of m types

$$\checkmark\ \underbrace{()(\{()\})([}_{b}\underbrace{])\{((([]()}_{b}\underbrace{)([])([\{\}]}_{b}\underbrace{())([])))\}}_{b}$$

Each block is D_m-consistent = is a substring of a string in D_m

☛ To ensure that at most ε-fraction of the blocs *are not D_m-consistent* (with constant probability):

- Select $\mathcal{O}(1/\varepsilon)$ blocks uniformly random;
- Read all b characters of each selected block and use a stack to check whether they are D_m-consistent.

Tester of Parnas et al. [PRR01]

D_m — well-balanced sequences of parentheses of m types

$$\checkmark\ \underbrace{()(\{()\})([}_{b}\underbrace{])\{((([]()}_{b}\underbrace{)([])([\{\}]}_{b}\underbrace{())([])))\}}_{b}$$

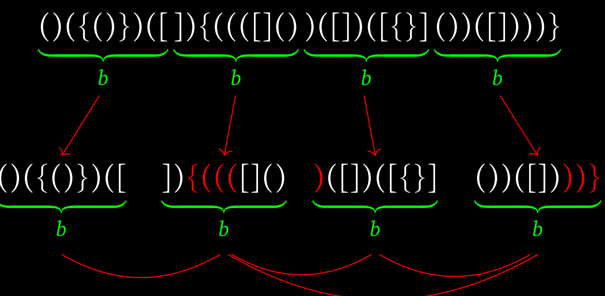
Each block is D_m-consistent = is a substring of a string in D_m

☛ To ensure that at most ε-fraction of the blocs *are not* D_m*-consistent* (with constant probability):

- Select $\mathcal{O}(1/\varepsilon)$ blocks uniformly random;
- Read all b characters of each selected block and use a stack to check whether they are D_m-consistent.

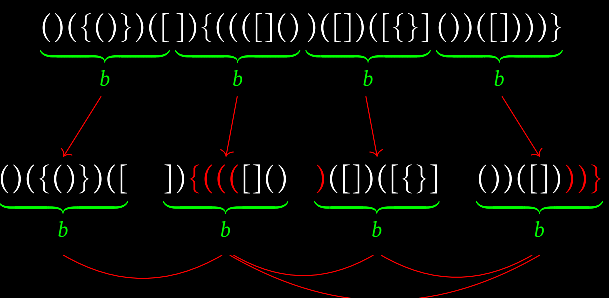
From now on, we can assume that the string is good *locally*. But can we guarantee that it is good *globally*?

Tester of Parnas et al. [PRR01]



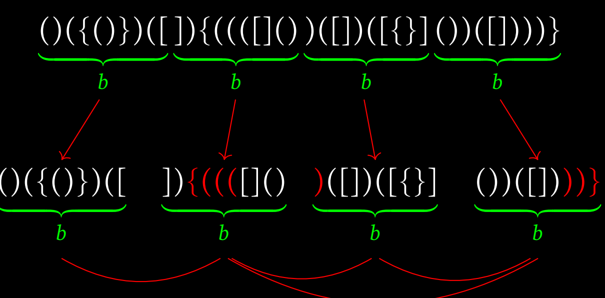
- **Matching graph:** Nodes = blocks, $(\text{block}_1, \text{block}_2)$ is an edge iff there is a parenthesis in block_1 that must be matched with a parenthesis in block_2

Tester of Parnas et al. [PRR01]



- **Approximate matching graph:** Nodes = blocks, $(\text{block}_1, \text{block}_2)$ is an edge iff there are *many* parentheses in block_1 that must be matched with parentheses in block_2

Tester of Parnas et al. [PRR01]



- Select $\mathcal{O}(1/\varepsilon)$ edges uniformly at random
- For each selected edge $(\text{block}_1, \text{block}_2)$ check if the excess parentheses in block_1 match with excess parentheses in block_2)
- **Main idea:** if there are many pairs of non-matching blocks, we will detect one of them with constant probability, and otherwise the string is almost well-balanced.

Building the approximate matching graph

$$])\{((([]() \Rightarrow \underbrace{))((((()()}_{\text{no types}} \Rightarrow \underbrace{(())((((()()))))}_{D_1} \Rightarrow \begin{cases} e_{closing}(S) = 2 \\ e_{opening}(S) = 4 \end{cases}$$

Excess parentheses in B_i that must be matched with excess parentheses in B_j:

$$\min\{e_{opening}(B_i), e_{closing}(B_{i+1}B_{i+2}\ldots B_j)\} - e_{closing}(B_{i+1}B_{i+2}\ldots B_{j-1})$$

$$\underbrace{()(\{()\})([}_{B_1} \quad \underbrace{])\{((([]()}_{B_2} \quad \underbrace{)([])([\{\}]}_{B_3} \quad \underbrace{())([])))\}}_{B_4}$$

Example: $\min\{e_{opening}(B_2), e_{closing}(B_3B_4)\} - e_{closing}(B_3) = 4 - 1 = 3$

Building the approximate matching graph

$$]\,)\{(((\,[\,]\,() \Rightarrow \underbrace{))((((()()}_{\text{no types}} \Rightarrow \underbrace{(())((((()())))}_{D_1} \Rightarrow \begin{cases} e_{closing}(S) = 4 \\ e_{opening}(S) = 2 \end{cases}$$

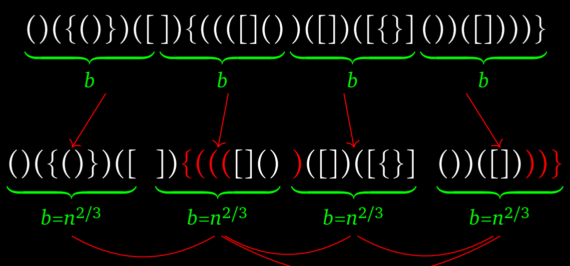
Observation (proof by induction)
$n_{closing}(X)$ = number of closing parentheses in X
$n_{opening}(X)$ = number of opening parentheses in X

$e_{closing}(S) = \max_{X-\text{prefix of } S}(n_{closing}(X) - n_{opening}(X))$

Lemma (Chernoff bound) By querying n^2/Δ^2 positions of a string S of length n, we can compute a Δ-additive approximation of $n_{closing}(X), n_{opening}(X)$ for any substring X of S correctly with constant probability.

In the tester, $\Delta = \varepsilon b$.

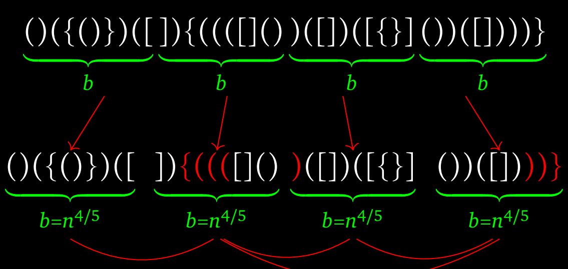
Tester of Parnas et al. [PRR01]



- If we omit the types in the input, the result $\in D_1$ — $\mathcal{O}(\mathbf{1})$
- Test that almost all the blocks are D_m-consistent — $\mathcal{O}(\mathbf{b})$
- Build an approximate matching graph — $\mathcal{O}(\mathbf{n^2/b^2})$
- Test that for almost all edges $(\text{block}_1, \text{block}_2)$ the excess parentheses of block_1, block_2 match — $\mathcal{O}(\mathbf{b})$

Total number of queries: $\mathcal{O}(\mathbf{b} + \mathbf{n^2/b^2}) = \mathcal{O}(\mathbf{n^{2/3}})$

New tester of Fischer et al. [FMS18]



- If we omit the types in the input, the result $\in D_1$ — $\mathcal{O}(\mathbf{1})$
- Test that almost all the blocks are D_m-consistent by applying the **new tester in a recursive fashion** — $\mathcal{O}(\mathbf{b}^{\mathbf{2/5}+\delta})$
- Build an approximate matching graph — $\mathcal{O}(\mathbf{n}^{\mathbf{2}}/\mathbf{b}^{\mathbf{2}})$
- Test that for almost all edges $(\text{block}_1, \text{block}_2)$ the excess parentheses of block_1, block_2 match using a *recursive inter-block matching procedure* — $\mathcal{O}(\mathbf{b}^{\mathbf{1/2}+\delta})$

Total number of queries: $\mathcal{O}(\mathbf{b}^{\mathbf{1/2}+\delta} + \mathbf{n}^{\mathbf{2}}/\mathbf{b}^{\mathbf{2}}) = \mathcal{O}(\mathbf{n}^{\mathbf{2/5}+\delta})$

Part III: Streaming property testing

Streaming property tester

stream

c

Input: a stream of characters

Output:

- Accept if the stream satisfies a property $\mathcal{P}$;
- Reject if the stream is far from any string that satisfies $\mathcal{P}$;
- Accept / reject otherwise.

Variations: Instead of asking if the whole stream satisfies $\mathcal{P}$, we can ask this question for each of its prefixes or for a n-length suffix of the current stream.

Streaming property tester

stream

c c

Input: a stream of characters

Output:

- Accept if the stream satisfies a property $\mathcal{P}$;
- Reject if the stream is far from any string that satisfies $\mathcal{P}$;
- Accept / reject otherwise.

Variations: Instead of asking if the whole stream satisfies $\mathcal{P}$, we can ask this question for each of its prefixes or for a n-length suffix of the current stream.

Streaming property tester

stream

c	c	a

Input: a stream of characters

Output:

- Accept if the stream satisfies a property $\mathcal{P}$;
- Reject if the stream is far from any string that satisfies $\mathcal{P}$;
- Accept / reject otherwise.

Variations: Instead of asking if the whole stream satisfies $\mathcal{P}$, we can ask this question for each of its prefixes or for a n-length suffix of the current stream.

Streaming property tester

stream

c c a a

Input: a stream of characters

Output:

- Accept if the stream satisfies a property $\mathcal{P}$;
- Reject if the stream is far from any string that satisfies $\mathcal{P}$;
- Accept / reject otherwise.

Variations: Instead of asking if the whole stream satisfies $\mathcal{P}$, we can ask this question for each of its prefixes or for a n-length suffix of the current stream.

Streaming property tester

stream

c c a a b

Input: a stream of characters

Output:

- Accept if the stream satisfies a property $\mathcal{P}$;
- Reject if the stream is far from any string that satisfies $\mathcal{P}$;
- Accept / reject otherwise.

Variations: Instead of asking if the whole stream satisfies $\mathcal{P}$, we can ask this question for each of its prefixes or for a n-length suffix of the current stream.

Streaming property tester

stream

c c a a b c

Input: a stream of characters

Output:

- Accept if the stream satisfies a property $\mathcal{P}$;
- Reject if the stream is far from any string that satisfies $\mathcal{P}$;
- Accept / reject otherwise.

Variations: Instead of asking if the whole stream satisfies $\mathcal{P}$, we can ask this question for each of its prefixes or for a n-length suffix of the current stream.

Streaming property tester

stream

c	c	a	a	b	c	...

Input: a stream of characters

Output:

- Accept if the stream satisfies a property $\mathcal{P}$;
- Reject if the stream is far from any string that satisfies $\mathcal{P}$;
- Accept / reject otherwise.

Variations: Instead of asking if the whole stream satisfies $\mathcal{P}$, we can ask this question for each of its prefixes or for a n-length suffix of the current stream.

Streaming property tester

stream

c	c	a	a	b	c	...	b

Input: a stream of characters

Output:

- Accept if the stream satisfies a property $\mathcal{P}$;
- Reject if the stream is far from any string that satisfies $\mathcal{P}$;
- Accept / reject otherwise.

Variations: Instead of asking if the whole stream satisfies $\mathcal{P}$, we can ask this question for each of its prefixes or for a n-length suffix of the current stream.

Streaming property tester

stream

c c a a b c … b c

Input: a stream of characters

Output:

- Accept if the stream satisfies a property $\mathcal{P}$;
- Reject if the stream is far from any string that satisfies $\mathcal{P}$;
- Accept / reject otherwise.

Variations: Instead of asking if the whole stream satisfies $\mathcal{P}$, we can ask this question for each of its prefixes or for a n-length suffix of the current stream.

Streaming property tester

stream

c c a a b c ... b c a

Input: a stream of characters

Output:

- Accept if the stream satisfies a property $\mathcal{P}$;
- Reject if the stream is far from any string that satisfies $\mathcal{P}$;
- Accept / reject otherwise.

Variations: Instead of asking if the whole stream satisfies $\mathcal{P}$, we can ask this question for each of its prefixes or for a n-length suffix of the current stream.

Streaming property tester

stream

c c a a b c ... b c a

accept/reject

Input: a stream of characters

Output:

- Accept if the stream satisfies a property $\mathcal{P}$;
- Reject if the stream is far from any string that satisfies $\mathcal{P}$;
- Accept / reject otherwise.

Variations: Instead of asking if the whole stream satisfies $\mathcal{P}$, we can ask this question for each of its prefixes or for a n-length suffix of the current stream.

Streaming property testers for formal languages

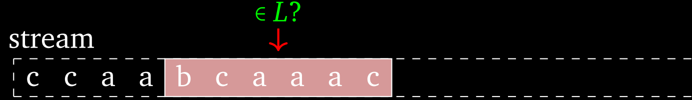
Visibly pushdown languages [FMdRS16]

- Read the whole stream of length n
- If the stream $\in$ visibly pushdown language L, ACCEPT
- If ED(stream, L) $\geq \varepsilon n$, REJECT
- S = $\mathcal{O}(\log^6 n \log(n/\delta)/\varepsilon^4)$, where δ is the error probability

Regular languages [GHLS19]

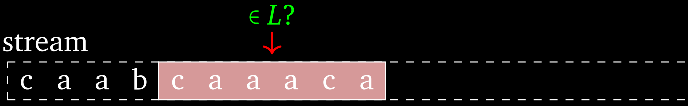
- When a new character arrives, ACCEPT if the n-length suffix of the current stream $\in$ regular language L
- REJECT if the Ham(suffix, L) $\geq \varepsilon n$
- Logarithmic or better space (details below)

Recognising regular languages



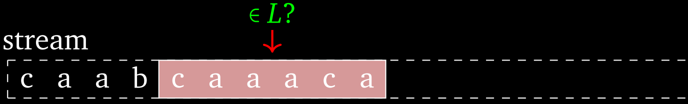
- **Streaming algorithm:** When a new character arrives, ACCEPT if the n-length suffix $\in L$, REJECT otherwise.
- **Streaming property tester:** When a new character arrives, ACCEPT if the n-length suffix $\in L$, REJECT if the distance between the suffix and L is at least εn.

Examples



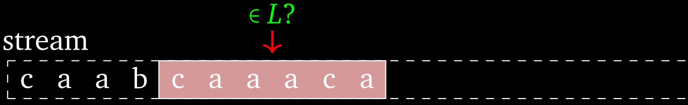
- $L_1 = \{w \in \{a, b, c\}^* : w \text{ ends with } b\}$

Examples



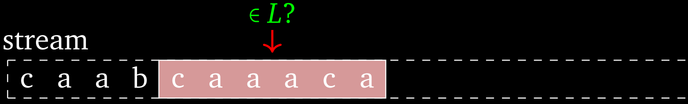
- $L_1 = \{w \in \{a,b,c\}^* : w \text{ ends with } b\}$
  $\mathcal{O}(1)$ bits of space: store the last character of the stream

Examples



- $L_1 = \{w \in \{a, b, c\}^* : w \text{ ends with } b\}$
  $\mathcal{O}(1)$ bits of space: store the last character of the stream
- $L_2 = \{w \in \{a, b, c\}^* : w \text{ contains } a\}$

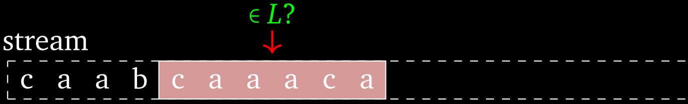
Examples



- $L_1 = \{w \in \{a,b,c\}^* : w \text{ ends with } b\}$
  $\mathcal{O}(1)$ bits of space: store the last character of the stream

- $L_2 = \{w \in \{a,b,c\}^* : w \text{ contains } a\}$
  $\mathcal{O}(\log n)$ bits of space: store the position of the last occurrence of a up to threshold n

Examples

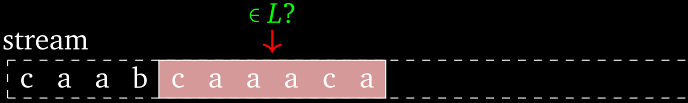


- $L_1 = \{w \in \{a,b,c\}^* : w \text{ ends with } b\}$
  $\mathcal{O}(1)$ bits of space: store the last character of the stream

- $L_2 = \{w \in \{a,b,c\}^* : w \text{ contains } a\}$
  $\mathcal{O}(\log n)$ bits of space: store the position of the last occurrence of a up to threshold n

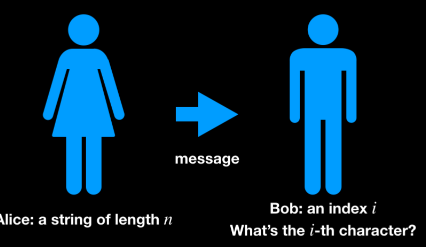
- $L_3 = \{w \in \{a,b,c\}^* : w \text{ starts with } a\}$

Examples



- $L_1 = \{w \in \{a,b,c\}^* : w \text{ ends with } b\}$
  $\mathcal{O}(1)$ bits of space: store the last character of the stream
- $L_2 = \{w \in \{a,b,c\}^* : w \text{ contains } a\}$
  $\mathcal{O}(\log n)$ bits of space: store the position of the last occurrence of a up to threshold n
- $L_3 = \{w \in \{a,b,c\}^* : w \text{ starts with } a\}$
  n bits by reduction to Index

Lower bound for L_3

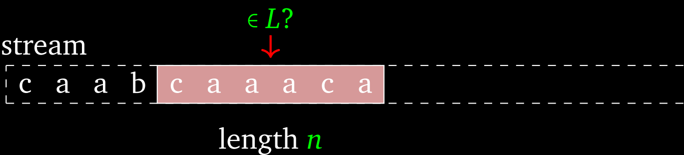


If there is a streaming algorithm for
$L_3 = \{w \in \{a, b, c\}^* : w \text{ starts with } a\}$, we can solve Index:

- 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 is in L_3, the i-th character is a

The algorithm must use n bits of space

Streaming algorithms



Space (bits)	Deterministic [GHK+18]	Randomised [GHL18a]
$\mathcal{O}(\mathbf{n})$	Reg	Reg
$\mathcal{O}(\log \mathbf{n})$	<LI, Len>	<LI, Len>
$\mathcal{O}(\log \log \mathbf{n})$	—	<ST, SF, Len>
$\mathcal{O}(\mathbf{1})$	<ST, Len>	<ST, Len>

Property tester for a regular language L [AKNS00]

- For each $1 \leq i \leq \log(c_1 \cdot \varepsilon)$, the tester samples $\frac{c_2 \cdot 2^{-i} \log^2(1/\varepsilon)}{\varepsilon}$ substrings of length 2^i at random.
- The constants c_1, c_2 depend *only* on the automaton for L.
- If Ham(input, L) $\geq \varepsilon n$, the tester REJECTS with prob. $\geq 3/4$. If the input $\in L$, the tester ACCEPTS with prob. 1.

---

[1] Dependency on ε not shown

Property tester for a regular language L [AKNS00]

- For each $1 \le i \le \log(c_1 \cdot \varepsilon)$, the tester samples $\frac{c_2 \cdot 2^{-i} \log^2(1/\varepsilon)}{\varepsilon}$ substrings of length 2^i at random.
- The constants c_1, c_2 depend *only* on the automaton for L.
- If Ham(input, L) $\ge \varepsilon n$, the tester REJECTS with prob. $\ge 3/4$. If the input $\in L$, the tester ACCEPTS with prob. 1.

Two-sided error streaming property tester

- Maintain a sample of random substrings of the current n-length suffix of the stream [BOZ12]
- For every character sampled, we pay $\mathcal{O}(\log n)$ bits of space. The sample can be smaller with probability $1/3$
- Space $= \mathcal{O}(\log n)$ bits [1]

[1] Dependency on ε not shown

Streaming property testers for reg. lang. [GHLS19]

Deterministic:

- Constant Hamming distance gap
- $\mathcal{O}(\log n)$ bits of space [2]

[2] Dependency on ε not shown

Streaming property testers for reg. lang. [GHLS19]

Deterministic:

- Constant Hamming distance gap
- $\mathcal{O}(\log n)$ bits of space [2]

One-sided error:

- Hamming distance gap $= \varepsilon n$
- $\mathcal{O}(\log n)$, $\mathcal{O}(\log \log n)$, or $\mathcal{O}(1)$ bits of space
- Language theoretic characterizations of these space classes

[2] Dependency on ε not shown

Streaming property testers for reg. lang. [GHLS19]

Deterministic:

- Constant Hamming distance gap
- $\mathcal{O}(\log n)$ bits of space [2]

One-sided error:

- Hamming distance gap $= \varepsilon n$
- $\mathcal{O}(\log n)$, $\mathcal{O}(\log \log n)$, or $\mathcal{O}(1)$ bits of space
- Language theoretic characterizations of these space classes

Two-sided error:

- Hamming distance gap $= \varepsilon n$
- $\mathcal{O}(1)$ bits

---

[2] Dependency on ε not shown

Summary of today's talk

Restricted-access models of computation for string processing:

- Streaming
- Property testing
- Streaming property testing

In many cases, considerable improvement in space, time, or both.

Open questions: New approaches to massive string processing? Limits of computation?

Thank you!

References I

- N. Alon, M. Krivelevich, I. Newman, and M. Szegedy, *Regular languages are testable with a constant number of queries*, SIAM J. Comput. **30** (2000), no. 6, 1842–1862.
- O. Ben-Eliezer, S. Korman, and D. Reichman, *Deleting and testing forbidden patterns in multi-dimensional arrays*, ICALP, LIPIcs, vol. 80, 2017, pp. 9:1–9:14.
- D. Breslauer and Z. Galil, *Real-time streaming string-matching*, ACM Trans. Algorithms **10** (2014), no. 4, 22:1–22:12.
- A. Babu, N. Limaye, J. Radhakrishnan, and G. Varma, *Streaming algorithms for language recognition problems*, Theor. Comput. Sci. **494** (2013), 13–23.

References II

- V. Braverman, R. Ostrovsky, and C. Zaniolo, *Optimal sampling from sliding windows*, J. Comput. Syst. Sci. **78** (2012), no. 1, 260–272.
- R. Clifford, A. Fontaine, E. Porat, B. Sach, and T. Starikovskaya, *Dictionary matching in a stream*, ESA, LNCS, vol. 9294, 2015, pp. 361–372.
- ———, *The k-mismatch problem revisited*, SODA, 2016.
- R. Clifford and T. Starikovskaya, *Approximate Hamming distance in a stream*, ICALP, vol. 55, 2016, pp. 20:1–20:14.
- F. Ergün, E. Grigorescu, E. Sadeqi Azer, and S. Zhou, *Streaming periodicity with mismatches*, APPROX, 2017, pp. 42:1–42:21.

References III

- ———, *Periodicity in data streams with wildcards*, CSR, 2018, pp. 90–105.
- F. Ergün, H. Jowhari, and M. Sağlam, *Periodicity in streams*, APPROX, 2010, pp. 545–559.
- N. François, F. Magniez, M. de Rougemont, and O. Serre, *Streaming property testing of visibly pushdown languages*, ESA, LIPIcs, vol. 57, 2016, pp. 43:1–43:17.
- E. Fischer, F. Magniez, and T. Starikovskaya, *Improved bounds for testing Dyck languages*, SODA, 2018, pp. 1529–1544.
- M. Ganardi, *Visibly pushdown languages over sliding windows*, STACS, LIPIcs, vol. 126, 2019, pp. 29:1–29:17.

References IV

- M. Ganardi, D. Hucke, D. König, M. Lohrey, and K. Mamouras, *Automata theory on sliding windows*, STACS, LIPIcs, vol. 96, 2018, pp. 31:1–31:14.
- M. Ganardi, D. Hucke, and M. Lohrey, *Querying regular languages over sliding windows*, FSTTCS, LIPIcs, vol. 65, 2016, pp. 18:1–18:14.
- ______, *Randomized sliding window algorithms for regular languages*, ICALP, LIPIcs, vol. 107, 2018, pp. 127:1–127:13.
- ______, *Sliding window algorithms for regular languages*, LATA, vol. 10792, 2018, pp. 26–35.
- M. Ganardi, D. Hucke, M. Lohrey, and T. Starikovskaya, *Sliding window property testing for regular languages*, ISAAC, LIPIcs, vol. 149, 2019, pp. 6:1–6:13.

References V

- M. Ganardi, A. Jez, and M. Lohrey, *Sliding windows over context-free languages*, MFCS, LIPIcs, vol. 117, 2018, pp. 15:1–15:15.

- P. Gawrychowski, O. Merkurev, A. M. Shur, and P. Uznanski, *Tight tradeoffs for real-time approximation of longest palindromes in streams*, Algorithmica **81** (2019), no. 9, 3630–3654.

- S. Golan and E. Porat, *Real-time streaming multi-pattern search for constant alphabet*, ESA, 2017, pp. 41:1–41:15.

- P. Gawrychowski, J. Radoszewski, and T. Starikovskaya, *Quasi-periodicity in streams*, CPM, LIPIcs, vol. 128, 2019, pp. 22:1–22:14.

References VI

- P. Gawrychowski and T. Starikovskaya, *Streaming dictionary matching with mismatches*, CPM, vol. 128, 2019, pp. 21:1–21:15.
- F. Magniez, C. Mathieu, and A. Nayak, *Recognizing well-parenthesized expressions in the streaming model*, SIAM J. Comput. **43** (2014), no. 6, 1880–1905.
- O. Merkurev and A. M. Shur, *Searching long repeats in streams*, CPM, LIPIcs, vol. 128, 2019, pp. 31:1–31:14.
- ———, *Searching runs in streams*, SPIRE, LNCS, vol. 11811, 2019, pp. 203–220.
- A. Ndione, A. Lemay, and J. Niehren, *Approximate membership for regular languages modulo the edit distance*, Theoretical Computer Science **487** (2013), 37–49.

References VII

- ______, *Sublinear DTD validity*, ICALP, 2015, pp. 739–751.
- B. Porat and E. Porat, *Exact and approximate pattern matching in the streaming model*, FOCS, 2009, pp. 315–323.
- M. Parnas, D. Ron, and R. Rubinfeld, *Testing parenthesis languages*, APPROX-RANDOM, LNCS, vol. 2129, 2001, pp. 261–272.
- T. Starikovskaya, *Communication and streaming complexity of approximate pattern matching*, CPM, 2017, pp. 13:1–13:11.