Fast Approximation Algorithms for Formal Languages

Gabriel Bathie

June 12, 2025

Scope: Robust techniques for extracting properties of text.

- *Properties of text:* **formal languages** (defined later), focus on *structure* instead of *meaning*
- *Techniques:* **algorithms**, for automated execution,
- *Robust:* to small changes in the text,

Strings: abstract model of text

Finite sequence of symbols from a finite set Σ.

English:	“`Azur is a nice cat`”	$\Sigma = \{\text{ASCII chars}\}$
DNA:	`CTTAGCACGACGATATTGTAACGCGTACT`	$\Sigma = \{\texttt{A, C, G, T}\}$
Robot movement:	$\uparrow\downarrow\rightarrow\leftarrow\leftarrow\uparrow\rightarrow\downarrow\downarrow\rightarrow\leftarrow\uparrow$	$\Sigma = \{\uparrow, \downarrow, \rightarrow, \leftarrow\}$

Does $P = \texttt{GTACGAAC}$ appear in

$T = \ldots\texttt{CTTAGCACGACGGGATATTGTACGAACGCGTACTAACA}\ldots$?

Does $P = \texttt{GTACGAAC}$ appear in

$T = \ldots\texttt{CTTAGCACGACGGGATATTGTACGAACGCGTACTAACA}\ldots$?

$\texttt{GTACGAAC}$

Does $P =$ `GTACGAAC` appear in

$$T = \ldots \texttt{CTTAGCACGACGGGATA?TGTACGAACGCGT??TAACA} \ldots ?$$

Corrupted data:
could be anything

With DNA:

- Sequencing Errors: 0.1% – 1% chance per symbol,
- Mutations: 10^{-7}% chance per symbol per year.

→ Search for fragments *similar* to P.

Does $P = \texttt{GTACGAAC}$ appear in

$$T = \ldots\texttt{CTTAGCACGACGGGATA?TGTACGAACGCGT??TAACA}\ldots?$$

$$\texttt{GTACGA}\cancel{\texttt{A}}\texttt{C} \qquad\qquad \texttt{GTACGAAC}$$

Mutations

With DNA:

- Sequencing Errors: 0.1% – 1% chance per symbol,
- Mutations: 10^{-7}% chance per symbol per year.

→ Search for fragments *similar* to P.

Terminology (I)

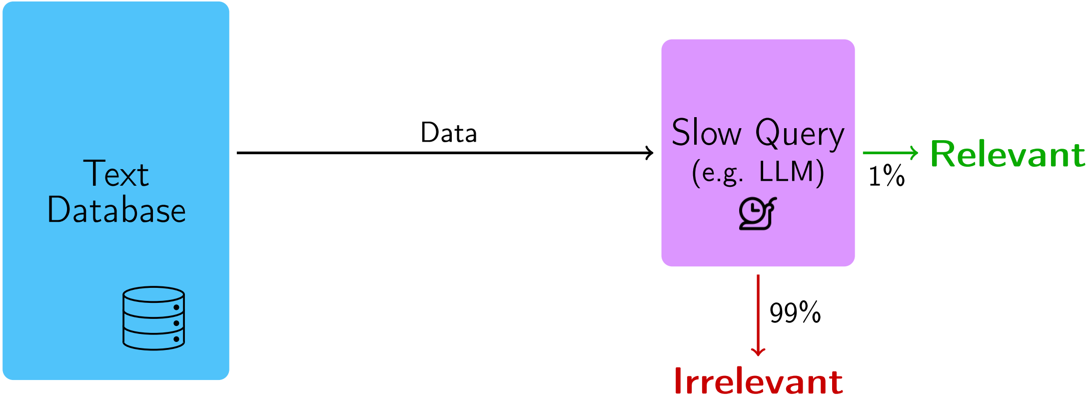
Pattern matching

Given two strings, P (the pattern) and T (the text), find all copies of P in T.

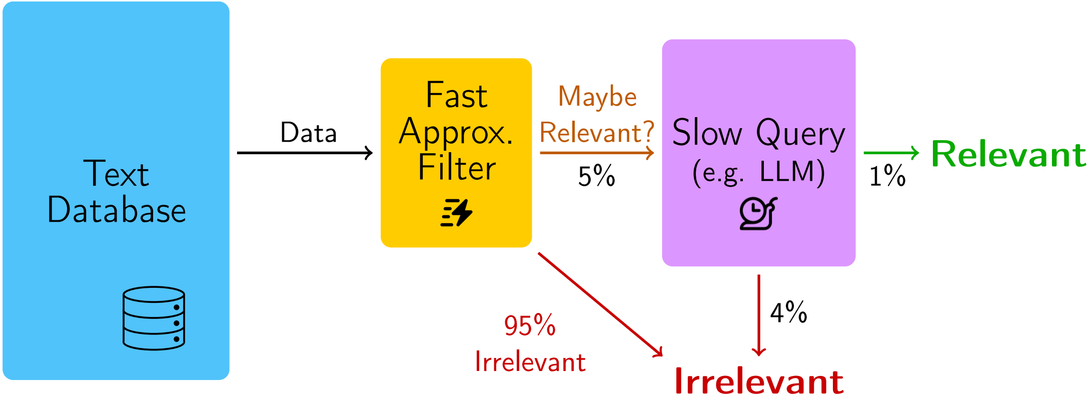
Robust version

Approximate Pattern matching

Given two strings P and T, find all substrings of T that are **similar** to P.


Text
Database
Data
Slow Query
(e.g. LLM)
1%
Relevant
99%
Irrelevant

Data mining



Text
Database
Data
Fast
Approx.
Filter
Maybe
Relevant?
5%
Slow Query
(e.g. LLM)
1%
Relevant
95%
Irrelevant
4%
Irrelevant

Terminology (II)

$$L_1 = \{\text{relevant strings}\} \qquad L_2 = \{S \text{ with an odd number of ``}A\text{''}\}$$

Formal Language

Set of strings.

$\rightarrow$ Models strings with a common property.

Language membership

Given a formal language L and a string S, is there a string in L that is equal to S?

Terminology (II)

$L_1 = \{\text{relevant strings}\}$ $\qquad$ $L_2 = \{S \text{ with an odd number of "}A\text{"}\}$

Formal Language

Set of strings.

$\rightarrow$ Models strings with a common property.

Approximate Language membership

Given a formal language L and a string S, is there a string in L that is **similar** to S?

Fast Approximation Algorithms for Formal Languages

Approximate Pattern Matching:
Does T have substrings *similar* to P?

Approximate Language Membership:
Are there strings *similar* to S in L?

Internal Pattern Matching in Small Space
with P. Charalampopoulos and T. Starikovskaya
CPM'24, **Best Paper Award**

Pattern Matching with Mismatches and Wildcards
with P. Charalampopoulos and T. Starikovskaya, ESA'24

Longest Common Extension with Wildcards
with P. Charalampopoulos and T. Starikovskaya, ESA'24

Property Testing of Regular Languages
with T. Starikovskaya, ICALP'21
with C. Mascle and N. Fijalkow, ICALP'25

Online Distance to Palindromes and Squares
with T. Kociumaka and T. Starikovskaya, ISAAC'23

Palindromic Length in Small Space
with J. Ellert and T. Starikovskaya (submitted)

$\rightarrow$ **Fast:** optimize *asymptotic worst-case* resource usage (time, memory, ...).

Fast Approximation Algorithms for Formal Languages

Approximate Pattern Matching:
Does T have substrings *similar* to P?

Approximate Language Membership:
Are there strings *similar* to S in L?

Internal Pattern Matching in Small Space
with P. Charalampopoulos and T. Starikovskaya
CPM'24, **Best Paper Award**

Pattern Matching with Mismatches and Wildcards
with P. Charalampopoulos and T. Starikovskaya, ESA'24

Longest Common Extension with Wildcards
with P. Charalampopoulos and T. Starikovskaya, ESA'24

Property Testing of Regular Languages
with T. Starikovskaya, ICALP'21
with C. Mascle and N. Fijalkow, ICALP'25

Online Distance to Palindromes and Squares
with T. Kociumaka and T. Starikovskaya, ISAAC'23

Palindromic Length in Small Space
with J. Ellert and T. Starikovskaya (submitted)

$\rightarrow$ **Fast:** optimize *asymptotic worst-case* resource usage (time, memory, ...).

Longest Common Extension (LCE)

$$LCE(i,j) = 4$$

$$\mathtt{A\underset{\substack{\uparrow\\ i}}{T}CTAGACTGGCATTAGATA\underset{\substack{\uparrow\\ j}}{T}CTATATTCCAG}$$

LCE is a key operation in Approximate Pattern Matching algorithms:

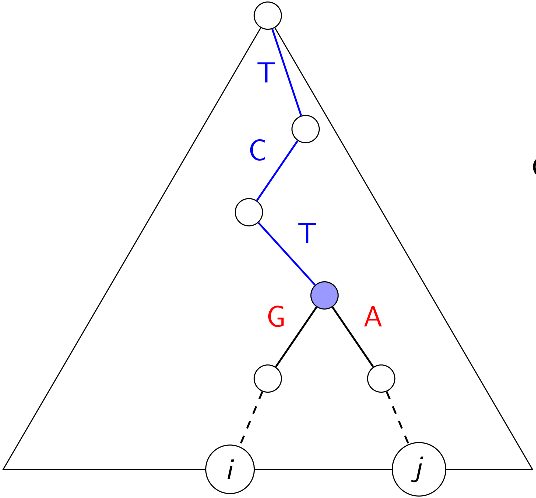
- [Landau and Vishkin, 1986]: PM with k edits $\simeq n \cdot k$ LCE queries,
- [Amir et al., 2004]: PM with k mismatches in $O(n\sqrt{k \log k})$ time with LCE.

Data structure for LCE

Build: given S, build a data structure $\mathcal{D}$,
Query: given i, j, compute $LCE(i,j)$ (using $\mathcal{D}$).

Data structure for LCE

Suffix Tree: Build $= O(n)$ time, Query $= O(1)$ time.



CGAATCTGCTAGCTTCTA...
$\uparrow$ i $\uparrow$ j

$LCE(i, j) = 3$

LCE with Wildcards (LCEW)

Wildcard

Special character “?” that matches all characters.

$$LCEW(i,j) = 6$$

$$\texttt{A}\underset{\substack{\uparrow\\ i}}{\texttt{T}}\texttt{CTA?ACTGGCATTAGATA}\underset{\substack{\uparrow\\ j}}{\texttt{T}}\texttt{CTATATTCCAG}$$

Observation:
Efficient LCEW data structure $\Rightarrow$ fast Approx. PM algorithms for string with wildcards.

Focus: data structure for LCEW.

Remark

The problem gets harder when we add more wildcards.

- 0 wildcard: normal LCE,
- ≥ 1 wildcards: normal LCE until you reach a wildcard:

$$\texttt{ATCTAGAC??GCATTA???ATCTATAT??CAG}$$

$$\uparrow \qquad\qquad\qquad\qquad\qquad \uparrow$$

“Right” parameter: G, number of *groups* of wildcards: $G = 3$ above.

Observation [Landau and Vishkin, 1986]

G groups of wildcards $\implies$ LCEW reduces to $G + 1$ LCE queries.

Build-query time trade-off

Observation [Landau and Vishkin, 1986]

G groups of wildcards $\Longrightarrow$ LCEW reduces to $G+1$ LCE queries.

Suffix Tree: Build $= O(n)$ time, Query $= O(1)$ time.

Data structure	Build time* p	Query time* q	Product $p \cdot q$
[Landau and Vishkin, 1986]	n	G	nG
[Crochemore et al., 2015]	nG	1	nG
[B., Charalampopoulos, and Starikovskaya, ESA'24]	nG/t	t	nG

*up to $\log^{O(1)} n$ factors.

Applications: approximate pattern matching

[Akutsu, 1995]

Approximate pattern matching with k edits reduces to $O(nk)$ LCEW queries.

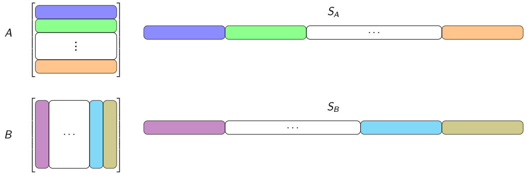
Algorithm	Time Complexity*
[Akutsu, 1995]	$n\sqrt{km}$
[Akutsu, 1995] + [Crochemore et al., 2015]	$nG + nk$
[B., Charalampopoulos, and Starikovskaya, ESA'24]	$n\sqrt{kG} + nk$

Lower bounds: Matrix Multiplication

- ▷ Product $p \cdot q = nG$ for all three data structures.
- ▷ Boolean Matrix Multiplication reduces to LCEW:

Boolean dot product:

$$A = (0, {\color{red}1}, 1, 0, 1) \to S_A = \texttt{?}{\color{red}\texttt{a}}\texttt{a?a}$$
$$B = (1, {\color{red}1}, 0, 0, 0) \to S_B = \texttt{b}{\color{red}\texttt{b}}\texttt{???}$$



Lower Bound

Combinatorial BMM is $\Omega(n^{3-\varepsilon}), \forall \varepsilon > 0$
$\implies p \cdot q = \Omega(n^{2-\varepsilon})$ when $G = \Theta(n)$.

Upper Bound: Sparse matrices $\to$ G small

Simple, combinatorial, deterministic sparse BMM in time $O(n\sqrt{\mathrm{nz}_{in} \cdot \mathrm{nz}_{out}})$.

- nz_{in} (nz_{out}): number of non-zero entries in input (output) matrices.
- Simple: $\sim$ 500 lines of Rust/C++.

Algorithm	Complexity
[Künnemann, 2018]	$O(\sqrt{\mathrm{nz}_{out}} \cdot n^2 + \mathrm{nz}_{out}^2)$
[Abboud et al., 2024]	$O(\mathrm{nz}_{in}\sqrt{\mathrm{nz}_{out}})$
[B., Charalampopoulos, and Starikovskaya, ESA'24]	$O(n\sqrt{\mathrm{nz}_{in} \cdot \mathrm{nz}_{out}})$

Table: Comparison with other sparse BMM algorithms.

Summary

- LCE is a key operation for approximate pattern matching.
- Data structure with build-query time trade-off for LCEW:

Build time*	$O(nG/t)$
Query time	$O(t)$

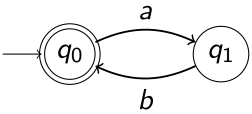
Table: Complexity, for $1 \leq t \leq G$

$\rightarrow$ Faster algorithm for pattern matching with wildcards and k edits,

$\rightarrow$ Connection to BMM: trade-off is optimal when $G = \Omega(n)$ (conditional),

$\rightarrow$ Reduction from BMM is sparse: combinatorial, deterministic algorithm for sparse BMM.

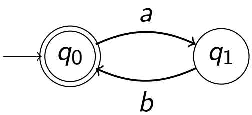
Property Testing



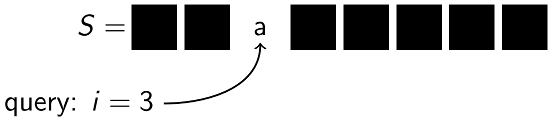
$L(\mathcal{A}) = (ab)^*$

$S =$ [redacted]

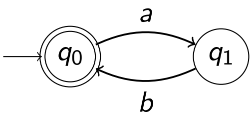
Question: is S in $L(\mathcal{A})$?
Goal: Minimize number of *queries*.
Algorithm: Randomness allowed, must be correct w.p. $\geq 2/3$.



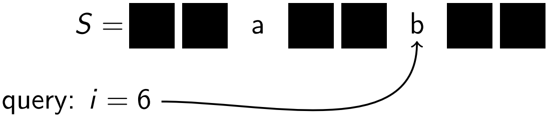
$L(\mathcal{A}) = (ab)^*$



Question: is S in $L(\mathcal{A})$?
Goal: Minimize number of *queries*.
Algorithm: Randomness allowed, must be correct w.p. $\geq 2/3$.

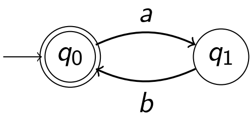


$L(\mathcal{A}) = (ab)^*$



Question: is S in $L(\mathcal{A})$?
Goal: Minimize number of *queries*.
Algorithm: Randomness allowed, must be correct w.p. $\geq 2/3$.

Property Testing

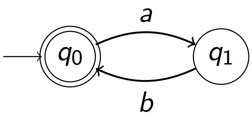


$L(\mathcal{A}) = (ab)^*$

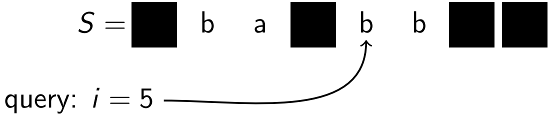
$S =$ [redacted] b a [redacted] b [redacted]

query: $i = 2$

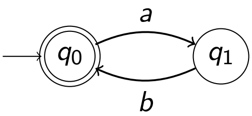
Question: is S in $L(\mathcal{A})$?
Goal: Minimize number of *queries*.
Algorithm: Randomness allowed, must be correct w.p. $\geq 2/3$.



$L(\mathcal{A}) = (ab)^*$



Question: is S in $L(\mathcal{A})$?
Goal: Minimize number of *queries*.
Algorithm: Randomness allowed, must be correct w.p. $\geq 2/3$.



$L(\mathcal{A}) = (ab)^*$

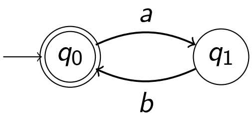
$S =$ [redacted] b a [redacted] b b [redacted]

Question: is S in $L(\mathcal{A})$?

Goal: Minimize number of *queries*. $\rightarrow \Omega(n)$ queries...

Algorithm: Randomness allowed, must be correct w.p. $\geq 2/3$.

Property Testing



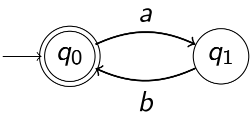
$L(\mathcal{A}) = (ab)^*$

$S =$ [redacted] b a [redacted] b b [redacted] [redacted]

Question: is S in $L(\mathcal{A})$ or ε-far from $L(\mathcal{A})$?
Goal: Minimize number of *queries*.
Algorithm: Randomness allowed, must be correct w.p. $\geq 2/3$.

- → ε-far from L: need to change $\geq \varepsilon n$ letters to be in L,
- → ε: parameter in $(0, 1)$, n: input length.

Property Testing



$L(\mathcal{A}) = (ab)^*$

$S =$ [redacted] b a [redacted] b b [redacted]

Question: is S in $L(\mathcal{A})$ or ε-far from $L(\mathcal{A})$?
Goal: Minimize number of *queries*.
Algorithm: Randomness allowed, must be correct w.p. $\geq 2/3$.

→ ε-far from L: need to change $\geq \varepsilon n$ letters to be in L,

→ ε: parameter in $(0, 1)$, n: input length.

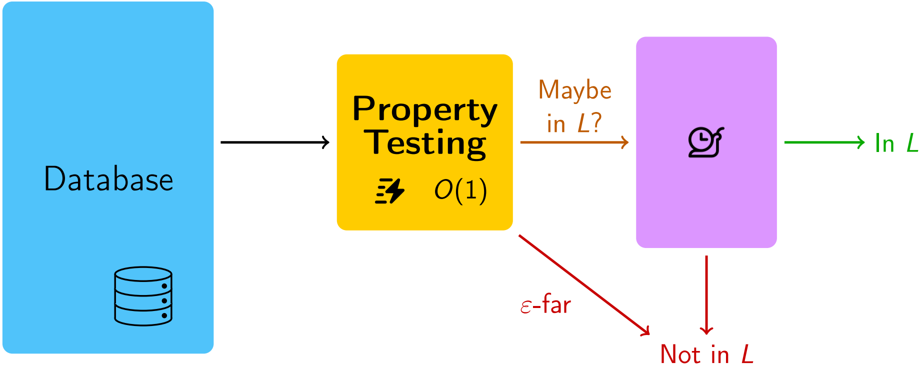
Theorem [Alon et al., 2001]

Algorithm with $O(\log^3(1/\varepsilon)/\varepsilon)$ queries, for any regular language.

Theorem [Alon et al., 2001]

Algorithm with $O(\log^3(1/\varepsilon)/\varepsilon)$ queries, for any regular language.

$\rightarrow$ Number of queries is *constant*, independent of input size!



Theorem [Alon et al., 2001]

- All regular languages: $O(\log^3(1/\varepsilon)/\varepsilon)$ queries.
- All "interesting" regular languages require $\Omega(1/\varepsilon)$ queries.

Theorem [B. and Starikovskaya, ICALP'21]

- All regular languages: $O(\log(1/\varepsilon)/\varepsilon)$ queries.
- There exists a regular language L_0 that requires $\Omega(\log(1/\varepsilon)/\varepsilon)$ queries.

$\rightarrow$ Problem closed?

Theorem [B. and Starikovskaya, ICALP'21]

There exists a regular language L_0 that requires $\Omega(\log(1/\varepsilon)/\varepsilon)$ queries.

$\rightarrow$ Applies to a single language.

There are languages with complexity $\Theta(1/\varepsilon)$, e.g. $L = a^*$ over $\Sigma = \{a, b\}$:

1. $S = aaaa \ldots aaa$ vs. S contains at least εn b's,
2. querying $O(1/\varepsilon)$ letters at random finds a "b" w.p. $\geq 2/3$.

There are "trivial" languages, that need 0 queries†, e.g. $L = a\Sigma^*$:
$\rightarrow$ Cannot be ε-far when n is large: can answer "in L".

†For large enough n.

Group by optimal query complexity:

- **"Hard":** $\Theta(\log(1/\varepsilon)/\varepsilon)$ queries $\to L_0$,
- **"Easy":** $\Theta(1/\varepsilon)$ queries $\to a^*$ $(\Sigma = \{a, b\})$,
- **"Trivial":** 0 queries $\to a\Sigma^*$, finite languages.

Question I

Are there other complexity classes?

Question II

Can we *characterize* the languages in each class?

Inspired by recent characterizations of:

- [Amarilli et al., 2021]: Dynamic Membership in Regular Languages,
- [Ganardi et al., 2024]: Regular Languages in Sliding Windows.

Property testing algorithms (I)

Testing $L = a^*$ over $\Sigma = \{a, b\}$:

1. $S = aaaa \ldots aaa$ vs. S contains at least εn b's,
2. querying $O(1/\varepsilon)$ letters at random finds a "b" w.p. $\geq 2/3$.

General case:

Blocking factor for L

String F such that if F appears in S, then S is not in L.

$\rightarrow$ for $L = a^*$, $F = b, abba, bbaba, \ldots,$ any word that contains a "b".

Lemma

If S if ε-far from L, then S contains $\Omega(\varepsilon n)$ non-overlapping blocking factors for L.

Property testing algorithms (II)

General case:

1. $S \in L \Rightarrow S$ contains no blocking factor,
2. S ε-far from $L \Rightarrow S$ contains $\Omega(\varepsilon n)$ blocking factors.

Algorithm:

- Sample random factors of S,
- If any is blocking, answer "far from L", otherwise "in L".

Theorem [B. and Starikovskaya, ICALP'21]

There is a sampling strategy that uses $O(\log(\varepsilon^{-1})/\varepsilon)$ queries.

Characterizing with (minimal) blocking factors

Minimal Blocking Factor (MBF) for L

Blocking factor F with no proper factor that is blocking for L.

$\rightarrow$ for $L = a^*$, $MBF(L) = \{b\}$: $ba, abba, bbaba, \ldots,$ are not minimal.

Trichotomy Theorem [B., Fijalkow, Mascle, ICALP'25]

Complexity is determined by the cardinality of the set of *minimal blocking factors*.

Class	Query Complexity ($\Theta(\cdot)$)	MBF(L)
Hard	$\log(1/\varepsilon)/\varepsilon$	Infinite
Easy	$1/\varepsilon$	Finite, non-empty
Trivial	0	$\emptyset$

Hidden details

- *strongly connected* automata = easy case,
- alphabet change: label letters with numbers.

$\rightarrow$ General case uses minimal blocking *sequences* ($\simeq$ sequences of MBF).

$\triangleright$ **Related Results** [B., Fijalkow, Mascle, ICALP'25]:

Structural

The set MBF(L) is a regular language.

Algorithmic

Given $\mathcal{A}$, classifying $L(\mathcal{A})$ is PSPACE-complete.

Summary

- Blocking factors: central to understanding property testing of regular languages.
- Query complexity is determined by the cardinality of the set of *minimal blocking sequences*.

Class	Query Complexity ($\Theta(\cdot)$)	MBS(L)
Hard	$\log(1/\varepsilon)/\varepsilon$	Infinite
Easy	$1/\varepsilon$	Finite, non-empty
Trivial	0	$\emptyset$

- Classification algorithm: given $\mathcal{A}$, classifying $L(\mathcal{A})$ is PSPACE-complete.

Summary

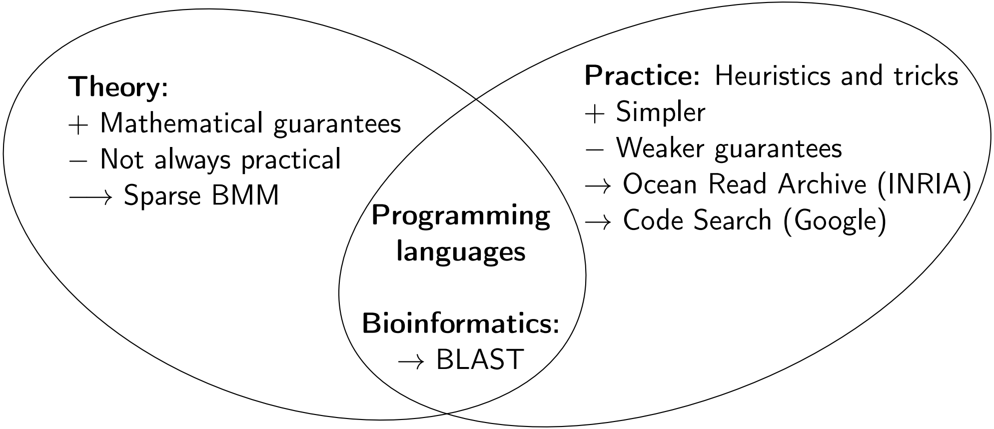
Also includes (not mentioned):

- Data structure for internal pattern matching in small-space [CPM'24, Best Paper],
- New algorithm for pattern matching with mismatches and wildcards [ESA'24],
- Small-space streaming algorithms for approx. language membership in palindromes/squares [ISAAC'23],
- Space-efficient algorithm for palindromic length (submitted).

Other work:

- Bolt (Software): fast LTL formula learning (submitted),
- Approximation scheme for Euclidean ultrametric embedding [AAAI'25],
- Constant delay enumeration of regular languages.

The gap between Theory and Practice



→ How can we make theoretical algorithms more practical?

→ Beyond worst-case frameworks for formal languages?

References I

Abboud, A., Bringmann, K., Fischer, N., and Künnemann, M. (2024).
The time complexity of fully sparse matrix multiplication.
In *Proc. of SODA*, pages 4670–4703.

Akutsu, T. (1995).
Approximate string matching with don't care characters.
Inf. Process. Lett., 55(5):235–239.

Alon, N., Krivelevich, M., Newman, I., and Szegedy, M. (2001).
Regular languages are testable with a constant number of queries.
SIAM Journal on Computing, 30(6):1842–1862.

Amarilli, A., Jachiet, L., and Paperman, C. (2021).
Dynamic Membership for Regular Languages.
In Bansal, N., Merelli, E., and Worrell, J., editors, *Proc. of ICALP'21*, volume 198 of *LIPIcs*, pages 116:1–116:17.

References II

Amir, A., Lewenstein, M., and Porat, E. (2004).
Faster algorithms for string matching with k mismatches.
J. Algorithms, 50(2):257–275.

B., G. and Starikovskaya, T. (2021).
Property Testing of Regular Languages with Applications to Streaming Property Testing of Visibly Pushdown Languages.
In *Proc. of ICALP*, volume 198 of *LIPIcs*, pages 119:1–119:17.

Borozdin, K., Kosolobov, D., Rubinchik, M., and Shur, A. M. (2017).
Palindromic Length in Linear Time.
In *Proc. of CPM*, volume 78 of *LIPIcs*, pages 23:1–23:12.

References III

Crochemore, M., Iliopoulos, C. S., Kociumaka, T., Kubica, M., Langiu, A., Radoszewski, J., Rytter, W., Szreder, B., and Waleń, T. (2015).
A note on the longest common compatible prefix problem for partial words.
Journal of Discrete Algorithms, 34:49–53.

François, N., Magniez, F., de Rougemont, M., and Serre, O. (2016).
Streaming property testing of visibly pushdown languages.
In *Proc. of ESA*, volume 57 of *LIPIcs*, pages 43:1–43:17.

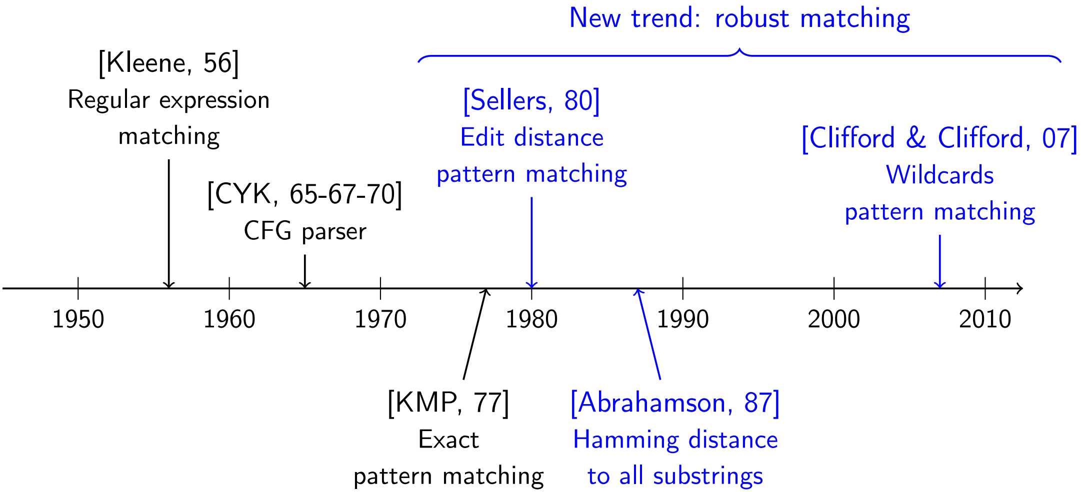
Ganardi, M., Hucke, D., Lohrey, M., Mamouras, K., and Starikovskaya, T. (2024).
Regular languages in the sliding window model.
CoRR, abs/2402.13385.

References IV

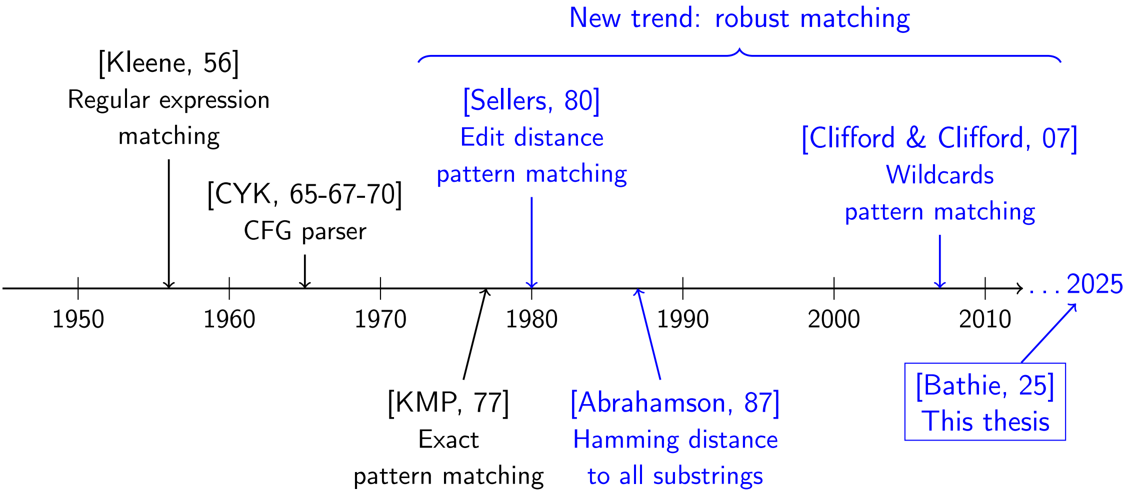
Künnemann, M. (2018).
On nondeterministic derandomization of Freivalds' algorithm: Consequences, avenues and algorithmic progress.
In *Proc. of ESA*, volume 112 of *LIPIcs*, pages 56:1–56:16.

Landau, G. M. and Vishkin, U. (1986).
Introducing efficient parallelism into approximate string matching and a new serial algorithm.
In *Proc. of STOC*, page 220–230.

Algorithms for text: a short history



Algorithms for text: a short history



Upper Bound: Sparse matrices $\to$ G small

Simple, combinatorial, deterministic sparse BMM in time $O(n\sqrt{\mathrm{nz}_{in} \cdot \mathrm{nz}_{out}})$.

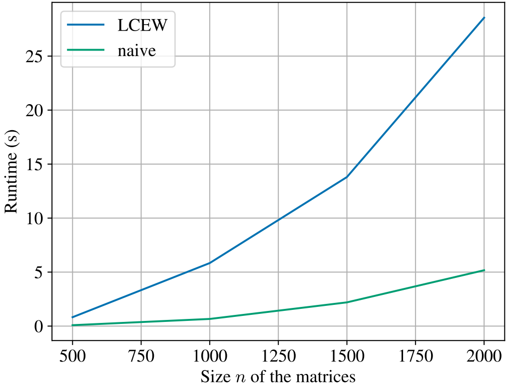


Figure: Algorithm is simple enough to be implemented in $\sim$ 500 lines of Rust.

Theorem [B. and Starikovskaya, ICALP'21]

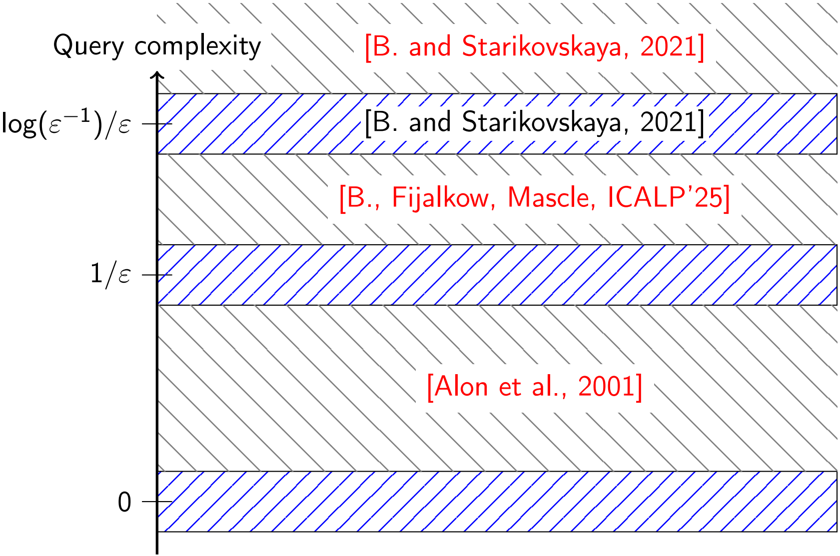
- All regular languages: $O(\log(1/\varepsilon)/\varepsilon)$ queries.
- There exists a regular language L_0 that requires $\Omega(\log(1/\varepsilon)/\varepsilon)$ queries.

Ideas

$\log^3(1/\varepsilon)/\varepsilon \to \log(1/\varepsilon)/\varepsilon$:

- Use ideas of [François et al., 2016],
- Tighter analysis of the sampling algorithm.

Language L_0: build hard input for our algorithm.


Query complexity
[B. and Starikovskaya, 2021]
$\log(\varepsilon^{-1})/\varepsilon$
[B. and Starikovskaya, 2021]
[B., Fijalkow, Mascle, ICALP'25]
$1/\varepsilon$
[Alon et al., 2001]
0

Original motivation

Lemma [François et al., 2016]

Streaming property testing (SPT) of VPLs reduces to multiple instances of property testing of regular languages.

Corollary [François et al., 2016]

SPT of VPLs can be solved using space $O(\log^6 n/\varepsilon^4)$.

Corollary [B. and Starikovskaya, ICALP'21]

SPT of VPLs can be solved using space $O(\log^5 n \log\log n/\varepsilon^3)$.

II-3 - Palindromic length in small space

Lemma [Borozdin et al., 2017]

Set of palindromic prefixes of $S \to$ union of $O(\log n)$ arithmetic progressions.

Arithmetic progression: string set of the form

$$\{AQ^i, i = 0, \ldots, t\}, \text{ e.g. } \{a, abc, abcbc, abcbcbc\}.$$

Theorem [B., Ellert and Starikovskaya, 2024]

Set of k-**palindromic prefixes** of $S \to$ union of $O(6^{k^2} \cdot \log^k n)$ affine sets of order k.

Affine set of order k: string set of the form $\{AQ_1^{i_1} \ldots Q_k^{i_k}, i_s = 0, \ldots, t_s, \forall s = 1, \ldots, k\}$.

$$S = \{a, ab, abb, abbb, acc, abcc, abbcc, abbbcc\} :$$
$$k = 2, A = a, Q_1 = b, t_1 = 3, Q_2 = cc, t_2 = 1.$$

II-3 - Palindromic length in small space

Theorem [Borozdin et al., 2017]

The palindromic length of S can be computed using $O(n)$ space and $O(n \log n)$ time.

Corollary [B., Ellert and Starikovskaya, 2024]

The palindromic length ℓ of S can be computed using $s = O(6^{\ell^2} \cdot \log^{\ell/2} n)$ space and $O(n \cdot s)$ time.