Sketch-based Approaches to Process Massive String Data

Garance Gourdel - Ph.D. Defense

26th of October 2023

Reviewers	Thierry LECROQ	Professor
	Simon J. PUGLISI	Professor
Jury members	Bastien CAZAUX	Assistant Professor
	Élise PRIEUR-GASTON	Assistant Professor
	Stéphane VIALETTE	Research Director
Supervisors	Pierre PETERLONGO	Research Director
	Tatiana STARIKOVSKAYA	Assistant Professor

Sketch-based approaches to process massive string data

Introduction

Contributions

1. Streaming Regular Expressions Membership
2. Pattern Matching for the Dynamic Time Warping Distance
3. Square Detection for Unordered Alphabets

Conclusion

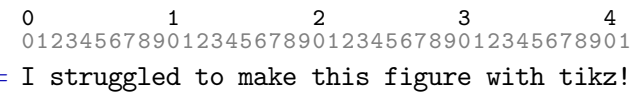
Sketch-based approaches to process massive **string data**

Sketch-based approaches to process massive **string data**

A string T of length n is a sequence $T[0]T[1]...T[n-1]$ of characters from a finite alphabet Σ of size σ.

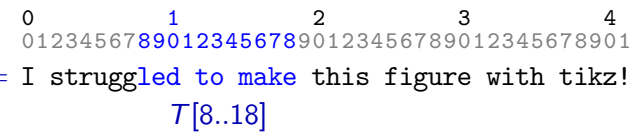
Sketch-based approaches to process massive **string data**

A string T of length n is a sequence $T[0]T[1]...T[n-1]$ of characters from a finite alphabet Σ of size σ.



Sketch-based approaches to process massive **string data**

A string T of length n is a sequence $T[0]T[1]...T[n-1]$ of characters from a finite alphabet Σ of size σ.



Definitions and Notations

The **substring** from position i to position j: $T[i]T[i+1]...T[j]$ is denoted $T[i..j]$.

Sketch-based approaches to process massive **string data**

A string T of length n is a sequence $T[0]T[1]...T[n-1]$ of characters from a finite alphabet Σ of size σ.

```
          0         1         2         3         4
          0123456789012345678901234567890123456789
T =       I struggled to make this figure with tikz!
```

$T[0..9]$ is a prefix

Definitions and Notations

The **substring** from position i to position j: $T[i]T[i+1]...T[j]$ is denoted $T[i..j]$.
Substrings of the form $T[0..j]$ are **prefixes**.

Sketch-based approaches to process massive **string data**

A string T of length n is a sequence $T[0]T[1]...T[n-1]$ of characters from a finite alphabet Σ of size σ.

```
          0         1         2         3         4
          0123456789012345678901234567890123456789012
T =       I struggled to make this figure with tikz!
```

$T[25..41]$ is a suffix

Definitions and Notations

The **substring** from position i to position j: $T[i]T[i+1]...T[j]$ is denoted $T[i..j]$.
Substrings of the form $T[0..j]$ are **prefixes**.
Substrings of the form $T[i..n-1]$ are **suffixes**.

Sketch-based approaches to process massive **string data**

A string T of length n is a sequence $T[0]T[1]...T[n-1]$ of characters from a finite alphabet Σ of size σ.

```
          0         1         2         3         4
          01234567890123456789012345678901234567890123
T =       I struggled to make this figure with tikz!
```

Definitions and Notations
The **substring** from position i to position j: $T[i]T[i+1]...T[j]$ is denoted $T[i..j]$.
Substrings of the form $T[0..j]$ are **prefixes**.
Substrings of the form $T[i..n-1]$ are **suffixes**.

In every algorithm in the presentation we assume the word RAM model with word of length logarithmic in the size of the input.

Sketch-based approaches to process massive **string data**

Strings are used in fields such as **Bioinformatics**, **Information Retrieval**, and **Cyber-security**.

Sketch-based approaches to process massive **string data**

Strings are used in fields such as **Bioinformatics**, **Information Retrieval**, and **Cyber-security**.

Strings	Binary file	DNA sequence	ASCII string file	UTF-8 string file
				
Alphabets	$\Sigma = \{0, 1\}$	$\Sigma = \{\mathtt{A}, \mathtt{T}, \mathtt{C}, \mathtt{G}\}$	$\Sigma = \{0, \ldots, 127\}$	$\Sigma = \{0, \ldots, 1\ 112\ 063\}$

Sketch-based approaches to process massive **string data**

Strings are used in fields such as **Bioinformatics**, **Information Retrieval**, and **Cyber-security**.

Strings	Binary file	DNA sequence	ASCII string file	UTF-8 string file
				
Alphabets	$\Sigma = \{0, 1\}$	$\Sigma = \{\mathtt{A}, \mathtt{T}, \mathtt{C}, \mathtt{G}\}$	$\Sigma = \{0, \ldots, 127\}$	$\Sigma = \{0, \ldots, 1\ 112\ 063\}$

Classical task:

Sketch-based approaches to process massive **string data**

Strings are used in fields such as **Bioinformatics**, **Information Retrieval**, and **Cyber-security**.

Strings	Binary file	DNA sequence	ASCII string file	UTF-8 string file
				
Alphabets	$\Sigma = \{0, 1\}$	$\Sigma = \{\texttt{A},\texttt{T},\texttt{C},\texttt{G}\}$	$\Sigma = \{0, \ldots, 127\}$	$\Sigma = \{0, \ldots, 1\ 112\ 063\}$

Classical task: Given a pattern $P = \texttt{th}$, does it occur in T and where?

```
0         1         2         3         4
01234567890123456789012345678901234567890123
```

$T =$ `I struggled to make this figure with tikz!`

Two occurences starting at positions **20** and **34** respectively.

Sketch-based approaches **to process** massive string data

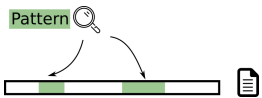
Sketch-based approaches **to process** massive string data

Three main types of tasks in this thesis:

Sketch-based approaches **to process** massive string data

Three main types of tasks in this thesis:

Complex matching



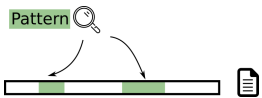
Examples:

- Regular expressions
- Wildcards
- Degenerate
- ...

Sketch-based approaches **to process** massive string data

Three main types of tasks in this thesis:

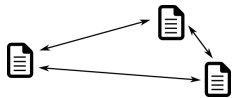
Complex matching



Examples:

- Regular expressions
- Wildcards
- Degenerate
- ...

Similarity measures and distances



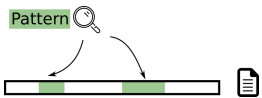
Examples:

- Hamming distance
- LCS
- Edit distance
- ...

Sketch-based approaches **to process** massive string data

Three main types of tasks in this thesis:

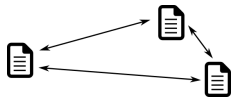
Complex matching



Examples:

- Regular expressions
- Wildcards
- Degenerate
- ...

Similarity measures and distances



Examples:

- Hamming distance
- LCS
- Edit distance
- ...

Repetitions detection

Examples:

- Periods
- Squares
- Runs
- ...

Sketch-based approaches to process **massive** string data

Strings are used in fields such as **Bioinformatics**, **Information Retrieval**, and **Cyber-security**.

Sketch-based approaches to process **massive** string data

Strings are used in fields such as **Bioinformatics**, **Information Retrieval**, and **Cyber-security**.

Data type	Webpage	E. coli genome	Software repository	Human genome
Size	$\sim$ 2 Mb	$\sim$ 5Mb	$<$ 1 Gb	$\sim$ 3 Gb

Sketch-based approaches to process **massive** string data

Strings are used in fields such as **Bioinformatics**, **Information Retrieval**, and **Cyber-security**.

Data type	Webpage	E. coli genome	Software repository	Human genome
Size	$\sim$ 2 Mb	$\sim$ 5Mb	$<$ 1 Gb	$\sim$ 3 Gb

Archive	Software Heritage	European Nucleotide (ENA)	Wayback Machine
Size	$>$ 1 Pb	$>$ 50 Pb	$>$ 70 Pb

Sketch-based approaches to process **massive** string data

Strings are used in fields such as **Bioinformatics**, **Information Retrieval**, and **Cyber-security**.

Data type	Webpage	E. coli genome	Software repository	Human genome
Size	$\sim$ 2 Mb	$\sim$ 5Mb	$<$ 1 Gb	$\sim$ 3 Gb

Archive	Software Heritage	European Nucleotide (ENA)	Wayback Machine
Size	$>$ 1 Pb	$>$ 50 Pb	$>$ 70 Pb

Many datasets of intermediate size: Tara ocean, 661K bacterial genome from ENA...

Sketch-based approaches to process **massive** string data

Strings are used in fields such as **Bioinformatics**, **Information Retrieval**, and **Cyber-security**.

Data type	Webpage	E. coli genome	Software repository	Human genome
Size	$\sim$ 2 Mb	$\sim$ 5Mb	$<$ 1 Gb	$\sim$ 3 Gb

Archive	Software Heritage	European Nucleotide (ENA)	Wayback Machine
Size	$>$ 1 Pb	$>$ 50 Pb	$>$ 70 Pb

Many datasets of intermediate size: Tara ocean, 661K bacterial genome from ENA...

Queries with a **high complexity** in time or space **cannot scale to large datasets**.

Sketch-based approaches to process **massive** string data

Strings are used in fields such as **Bioinformatics**, **Information Retrieval**, and **Cyber-security**.

Data type	Webpage	E. coli genome	Software repository	Human genome
Size	$\sim$ 2 Mb	$\sim$ 5Mb	$<$ 1 Gb	$\sim$ 3 Gb

Archive	Software Heritage	European Nucleotide (ENA)	Wayback Machine
Size	$>$ 1 Pb	$>$ 50 Pb	$>$ 70 Pb

Many datasets of intermediate size: Tara ocean, 661K bacterial genome from ENA...

Queries with a **high complexity** in time or space **cannot scale to large datasets**.

But some part of the data can be **redundant** or **not needed** for the queries we want?

Sketch-based approaches to process **massive** string data

Strings are used in fields such as **Bioinformatics**, **Information Retrieval**, and **Cyber-security**.

Data type	Webpage	E. coli genome	Software repository	Human genome
Size	$\sim$ 2 Mb	$\sim$ 5Mb	$<$ 1 Gb	$\sim$ 3 Gb

Archive	Software Heritage	European Nucleotide (ENA)	Wayback Machine
Size	$>$ 1 Pb	$>$ 50 Pb	$>$ 70 Pb

Many datasets of intermediate size: Tara ocean, 661K bacterial genome from ENA...

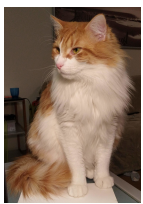
Queries with a **high complexity** in time or space **cannot scale to large datasets**.

But some part of the data can be **redundant** or **not needed** for the queries we want?

Sketch-based approaches to process massive string data

Sketch-based approaches to process massive string data

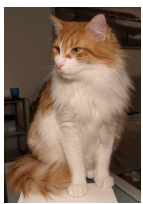
A **sketch** is a **lossless** or **lossy** compression that **keeps only the essential characteristic of the input** needed to answer a specified type of query.



"Is this a cat?"

Sketch-based approaches to process massive string data

A **sketch** is a **lossless** or **lossy** compression that **keeps only the essential characteristic of the input** needed to answer a specified type of query.



"Is this a cat?"

Examples:

Lossy: Karp–Rabin fingerprints

Lossless: Lempel–Ziv factorization

Example of lossy sketch: Karp–Rabin fingerprints

Example of lossy sketch: Karp–Rabin fingerprints

A hash function φ such that,
for two string X and Y with $|X| = |Y|$:

- If $X = Y$, then $\varphi(X) = \varphi(Y)$.
- If $\varphi(X) = \varphi(Y)$ then the strings match w.h.p. and it can be tested in $\mathcal{O}(1)$ time.

Karp–Rabin fingerprints

For $p > |\Sigma|$ a prime and $b > |\Sigma|$, the fingerprint of a string P of length m is:

$$\varphi_{p,b}(P) = \sum_{i=0}^{m-1} P[i] b^{m-i-1} \bmod p$$

Example of lossy sketch: Karp–Rabin fingerprints

A hash function φ such that,
for two string X and Y with $|X| = |Y|$:

- If $X = Y$, then $\varphi(X) = \varphi(Y)$.
- If $\varphi(X) = \varphi(Y)$ then the strings match w.h.p. and it can be tested in $\mathcal{O}(1)$ time.

Karp–Rabin fingerprints

For $p > |\Sigma|$ a prime and $b > |\Sigma|$, the fingerprint of a string P of length m is:

$$\varphi_{p,b}(P) = \sum_{i=0}^{m-1} P[i]b^{m-i-1} \bmod p$$

Occupies constant space, but we cannot reconstruct the original strings.

Example of lossy sketch: Karp–Rabin fingerprints

A hash function φ such that,
for two string X and Y with $|X| = |Y|$:

- If $X = Y$, then $\varphi(X) = \varphi(Y)$.
- If $\varphi(X) = \varphi(Y)$ then the strings match w.h.p. and it can be tested in $\mathcal{O}(1)$ time.

Karp–Rabin fingerprints

For $p > |\Sigma|$ a prime and $b > |\Sigma|$, the fingerprint of a string P of length m is:

$$\varphi_{p,b}(P) = \sum_{i=0}^{m-1} P[i]b^{m-i-1} \bmod p$$

Occupies constant space, but we cannot reconstruct the original strings.

Given X, we can compute $\varphi(X)$ in $\mathcal{O}(|X|)$ time.

Example of lossy sketch: Karp–Rabin fingerprints

A hash function φ such that,
for two string X and Y with $|X| = |Y|$:

- If $X = Y$, then $\varphi(X) = \varphi(Y)$.
- If $\varphi(X) = \varphi(Y)$ then the strings match w.h.p. and it can be tested in $\mathcal{O}(1)$ time.

Karp–Rabin fingerprints

For $p > |\Sigma|$ a prime and $b > |\Sigma|$, the fingerprint of a string P of length m is:

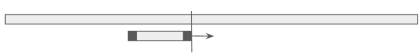
$$\varphi_{p,b}(P) = \sum_{i=0}^{m-1} P[i]b^{m-i-1} \bmod p$$

Occupies constant space, but we cannot reconstruct the original strings.

Given X, we can compute $\varphi(X)$ in $\mathcal{O}(|X|)$ time.

Rolling hash function

For a string T and $i < j$, given $\varphi(T[i..j])$, $T[i]$ and $T[j+1]$, we can compute $\varphi(T[i+1..j+1])$ in $\mathcal{O}(1)$ time.



Example of lossless sketch: Lempel–Ziv factorization

Decomposition in phrases: $T = f_1 f_2 \dots f_z$, (LZ76 with overlap version)

Example of lossless sketch: Lempel–Ziv factorization

Decomposition in phrases: $T = f_1 f_2 \dots f_z$, (LZ76 with overlap version)

	f_1	f_2	f_3	f_4		f_5			f_6					f_7							f_8		f_9		
$T =$	a	b	c	b	a	b	a	a	b	c	b	a	c	b	a	b	a	a	b	c	z	z	b	a	b	a

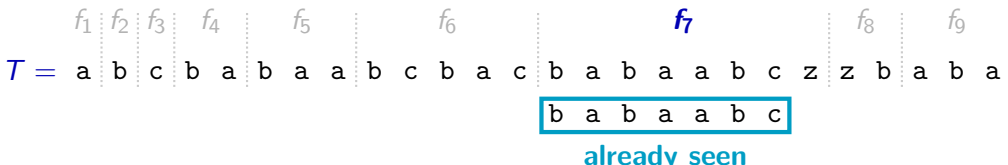
Example of lossless sketch: Lempel–Ziv factorization

Decomposition in phrases: $T = f_1 f_2 \dots f_z$, (LZ76 with overlap version)
The unique LZ-phrase starting at position s, $f_i = T[s..e]$ is such that $T[s..e]$ is the longest prefix of $T[s..n-1]$ that has a previous occurrence in $T[0..e]$.

	f_1	f_2	f_3	f_4	f_5	f_6	$\boldsymbol{f_7}$	f_8	f_9
$T =$	a	b	c	b a	b a a	b c b a c	b a b a a b c z	z b	a b a

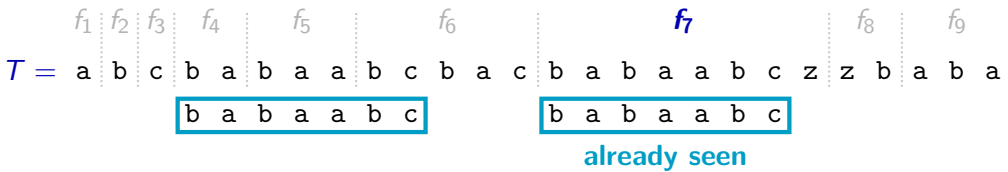
Example of lossless sketch: Lempel–Ziv factorization

Decomposition in phrases: $T = f_1 f_2 \dots f_z$, (LZ76 with overlap version)
The unique LZ-phrase starting at position s, $f_i = T[s..e]$ is such that $T[s..e]$ is the longest prefix of $T[s..n-1]$ that has a previous occurrence in $T[0..e]$.



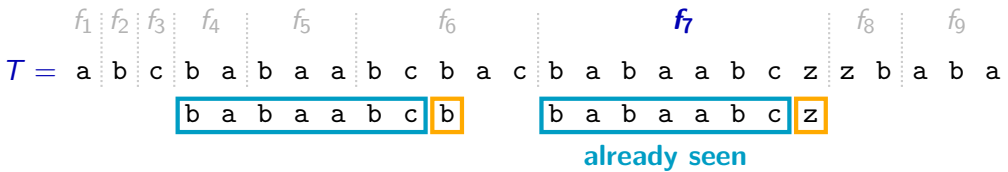
Example of lossless sketch: Lempel–Ziv factorization

Decomposition in phrases: $T = f_1 f_2 \dots f_z$, (LZ76 with overlap version)
The unique LZ-phrase starting at position s, $f_i = T[s..e]$ is such that $T[s..e]$ is the longest prefix of $T[s..n-1]$ that has a previous occurrence in $T[0..e]$.



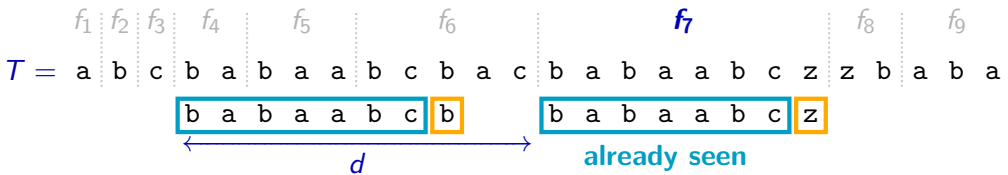
Example of lossless sketch: Lempel–Ziv factorization

Decomposition in phrases: $T = f_1 f_2 \dots f_z$, (LZ76 with overlap version)
The unique LZ-phrase starting at position s, $f_i = T[s..e]$ is such that $T[s..e]$ is the longest prefix of $T[s..n-1]$ that has a previous occurrence in $T[0..e]$.



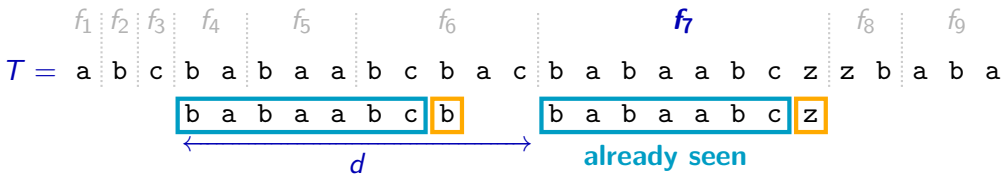
Example of lossless sketch: Lempel–Ziv factorization

Decomposition in phrases: $T = f_1 f_2 \dots f_z$, (LZ76 with overlap version)
The unique LZ-phrase starting at position s, $f_i = T[s..e]$ is such that $T[s..e]$ is the longest prefix of $T[s..n-1]$ that has a previous occurrence in $T[0..e]$.



Example of lossless sketch: Lempel–Ziv factorization

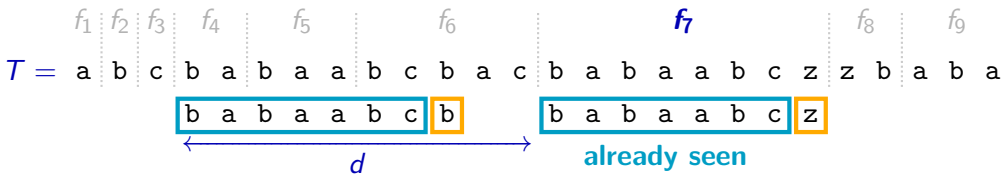
Decomposition in phrases: $T = f_1 f_2 \dots f_z$, (LZ76 with overlap version)
The unique LZ-phrase starting at position s, $f_i = T[s..e]$ is such that $T[s..e]$ is the longest prefix of $T[s..n-1]$ that has a previous occurrence in $T[0..e]$.



Used for **compression** by representing $f_i = T[s..e]$ by $(d, e-s, T[e])$.

Example of lossless sketch: Lempel–Ziv factorization

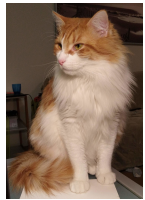
Decomposition in phrases: $T = f_1 f_2 \dots f_z$, (LZ76 with overlap version)
The unique LZ-phrase starting at position s, $f_i = T[s..e]$ is such that $T[s..e]$ is the longest prefix of $T[s..n-1]$ that has a previous occurrence in $T[0..e]$.



Used for **compression** by representing $f_i = T[s..e]$ by $(d, e-s, T[e])$. In the worst case, no compression, but very efficient in practice.

What is a sketch?

A **sketch** is a **lossless** or **lossy** compression that **keeps only the essential characteristic of the input** needed to answer a specified type of query.



"Is this a cat?"

Examples:

Lossy: Karp–Rabin fingerprints

- They occupy constant space,
- can check whether two strings match with high probability,
- but cannot be used to reconstruct the original string.

Lossless: Lempel–Ziv factorization

- It is a very efficient compression in practice (used in `.png` or `.zip`),
- can reconstruct the original string,
- but in the worst case it does not compress.

Why and how do we use sketches?

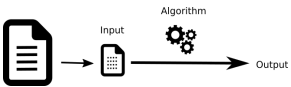
Sketches are typically much **smaller**, thus are promising for **scaling to larger** datasets.

Why and how do we use sketches?

Sketches are typically much **smaller**, thus are promising for **scaling to larger** datasets.

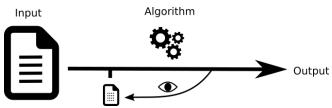
Two main approaches in this thesis:

Sketch as input



Operating directly on the sketch given as input (not decompressing).

Sketch as you go



Compute the sketch of the input on the fly and use it later on in your algorithm.

- For streaming
- For approximation

Contributions

LCS with approximatly k misimatches

CPM'20

XBWT indexing of readsets

WABI'21

Streaming regular expression

SODA'22

Pattern matching DTW

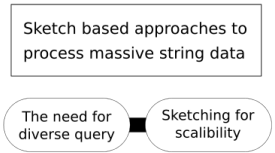
SPIRE'22

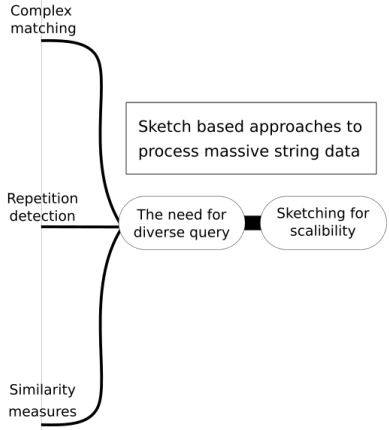
Squares for unoredered alphabets

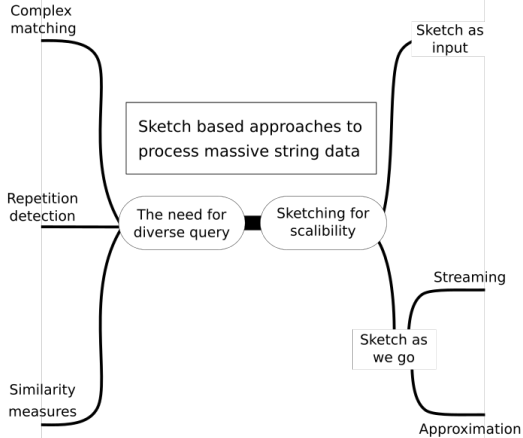
SODA'23

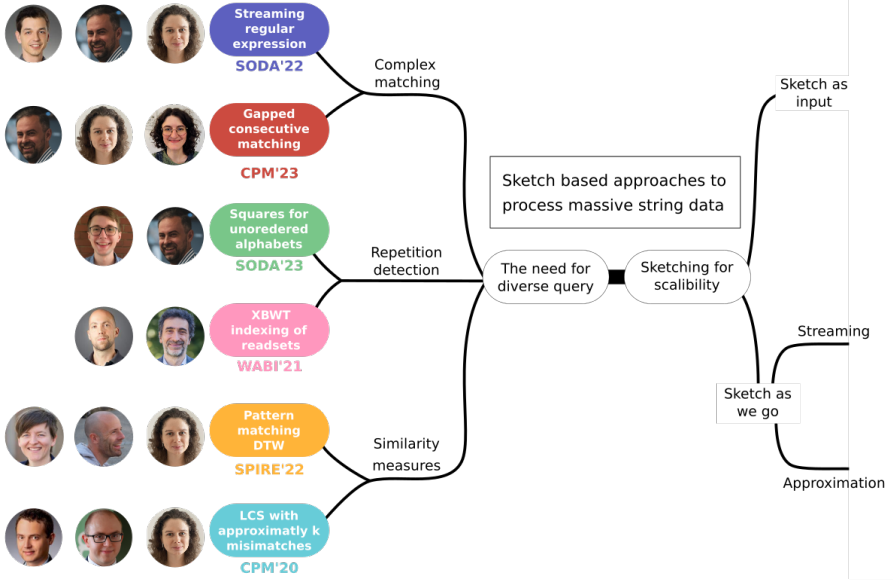
Gapped consecutive matching

CPM'23


Sketch based approaches to process massive string data
The need for diverse query
Sketching for scalibility


Complex matching
Sketch based approaches to process massive string data
Repetition detection
The need for diverse query
Sketching for scalibility
Similarity measures


Complex matching
Repetition detection
Similarity measures
The need for diverse query
Sketch based approaches to process massive string data
Sketching for scalibility
Sketch as input
Streaming
Sketch as we go
Approximation


Sketch based approaches to process massive string data
The need for diverse query
Sketching for scalibility
Sketch as input
Sketch as we go
Streaming
Approximation
Complex matching
Repetition detection
Similarity measures
Streaming regular expression
SODA'22
Gapped consecutive matching
CPM'23
Squares for unoredered alphabets
SODA'23
XBWT indexing of readsets
WABI'21
Pattern matching DTW
SPIRE'22
LCS with approximatly k misimatches
CPM'20




Streaming regular expression
SODA'22
Complex matching



Gapped consecutive matching
CPM'23


Squares for unoredered alphabets
SODA'23
Repetition detection

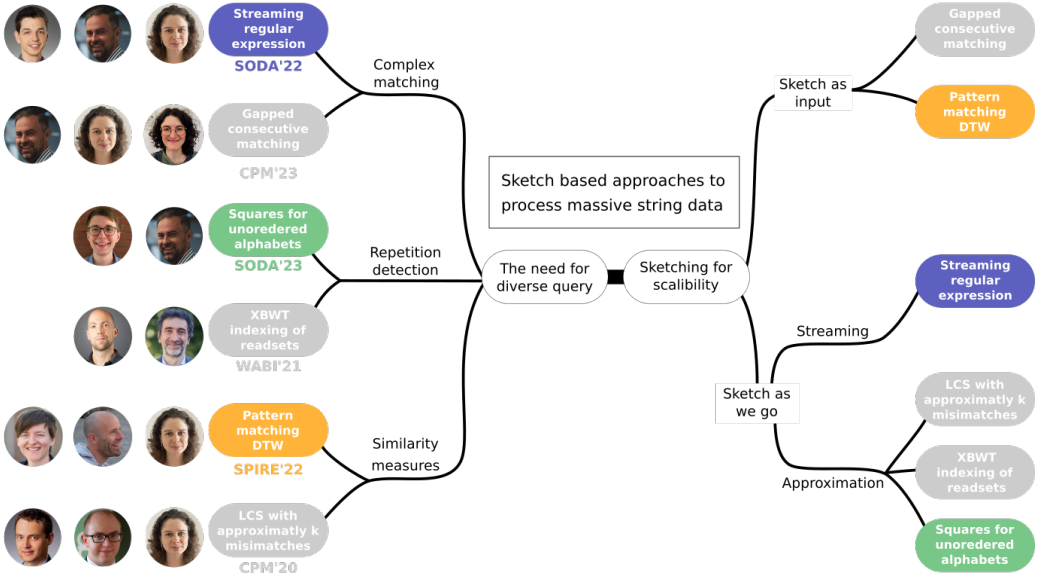

XBWT indexing of readsets
WABI'21



Pattern matching DTW
SPIRE'22
Similarity measures



LCS with approximatly k misimatches
CPM'20
Sketch based approaches to process massive string data
The need for diverse query
Sketching for scalibility
Sketch as input
Gapped consecutive matching
Pattern matching DTW
Sketch as we go
Streaming
Streaming regular expression
Approximation
LCS with approximatly k misimatches
XBWT indexing of readsets
Squares for unoredered alphabets


Sketch based approaches to process massive string data
The need for diverse query
Sketching for scalibility
Complex matching
Streaming regular expression
SODA'22
Gapped consecutive matching
CPM'23
Repetition detection
Squares for unoredered alphabets
SODA'23
XBWT indexing of readsets
WABI'21
Similarity measures
Pattern matching DTW
SPIRE'22
LCS with approximatly k misimatches
CPM'20
Sketch as input
Gapped consecutive matching
Pattern matching DTW
Sketch as we go
Streaming
Streaming regular expression
Approximation
LCS with approximatly k misimatches
XBWT indexing of readsets
Squares for unoredered alphabets

1st Contribution

Streaming Regular Expression Membership and Pattern Matching

SODA'22



Bartłomiej Dudek, Paweł Gawrychowski, Tatiana Starikovskaya

Regular expressions

Regular expressions (regexp)

either a character from Σ or recursively defined from other regular expressions R_1 and R_2:

1. $R_1 \cdot R_2$ (concatenation),
2. $R_1 | R_2$ (union),
3. R_1^* (Kleene star).

Regular expressions

Regular expressions (regexp)

either a character from Σ or recursively defined from other regular expressions R_1 and R_2:

1. $R_1 \cdot R_2$ (concatenation),
2. $R_1|R_2$ (union),
3. R_1^* (Kleene star).

Ex: $b(b|ab)^*ab$

✓ $bbbbbabab$
✓ $bbbabbbbab$
✗ $bbbaabbbab$
✗ $baba$
✗ $abab$

Regular expressions

Regular expressions (regexp)

either a character from Σ or recursively defined from other regular expressions R_1 and R_2:

1. $R_1 \cdot R_2$ (concatenation),
2. $R_1|R_2$ (union),
3. R_1^* (Kleene star).

Ex: $b(b|ab)^*ab$

- ✓ *bbbbbabab*
- ✓ *bbbabbbbab*
- ✗ *bbbaabbbab*
- ✗ *baba*
- ✗ *abab*

Used in databases,

Regular expressions

Regular expressions (regexp)

either a character from Σ or recursively defined from other regular expressions R_1 and R_2:

1. $R_1 \cdot R_2$ (concatenation),
2. $R_1 | R_2$ (union),
3. R_1^* (Kleene star).

Ex: $b(b|ab)^*ab$

- ✓ *bbbbbabab*
- ✓ *bbbabbbbab*
- ✗ *bbbaabbbab*
- ✗ *baba*
- ✗ *abab*

Used in databases, data mining,

Regular expressions

Regular expressions (regexp)

either a character from Σ or recursively defined from other regular expressions R_1 and R_2:

1. $R_1 \cdot R_2$ (concatenation),
2. $R_1|R_2$ (union),
3. R_1^* (Kleene star).

Ex: $b(b|ab)^*ab$

- ✓ *bbbbbabab*
- ✓ *bbbabbbbab*
- ✗ *bbbaabbbab*
- ✗ *baba*
- ✗ *abab*

Used in databases, data mining, secret detection,

Regular expressions

Regular expressions (regexp)

either a character from Σ or recursively defined from other regular expressions R_1 and R_2:

1. $R_1 \cdot R_2$ (concatenation),
2. $R_1|R_2$ (union),
3. R_1^* (Kleene star).

Ex: $b(b|ab)^*ab$

- ✓ *bbbbbabab*
- ✓ *bbbabbbbab*
- ✗ *bbbaabbbab*
- ✗ *baba*
- ✗ *abab*

Used in databases, data mining, secret detection, computer networks,

Regular expressions

Regular expressions (regexp)

either a character from Σ or recursively defined from other regular expressions R_1 and R_2:

1. $R_1 \cdot R_2$ (concatenation),
2. $R_1|R_2$ (union),
3. R_1^* (Kleene star).

Ex: $b(b|ab)^*ab$

✓ *bbbbbabab*
✓ *bbbabbbbab*
✗ *bbbaabbbab*
✗ *baba*
✗ *abab*

Used in databases, data mining, secret detection, computer networks, protein search

Regular expressions

Regular expressions (regexp)

either a character from Σ or recursively defined from other regular expressions R_1 and R_2:

1. $R_1 \cdot R_2$ (concatenation),
2. $R_1|R_2$ (union),
3. R_1^* (Kleene star).

Ex: $b(b|ab)^*ab$

✓ *bbbbbabab*
✓ *bbbabbbbab*
✗ *bbbaabbbab*
✗ *baba*
✗ *abab*

Used in databases, data mining, secret detection, computer networks, protein search...

Regular expressions

Regular expressions (regexp)

either a character from Σ or recursively defined from other regular expressions R_1 and R_2:

1. $R_1 \cdot R_2$ (concatenation),
2. $R_1|R_2$ (union),
3. R_1^* (Kleene star).

Ex: $b(b|ab)^*ab$

- ✓ *bbbbbabab*
- ✓ *bbbabbbbab*
- ✗ *bbbaabbbab*
- ✗ *baba*
- ✗ *abab*

Used in databases, data mining, secret detection, computer networks, protein search...

Given a regular expression R and a string T, **membership** is: check if R matches T.

Regular expressions

Regular expressions (regexp)

either a character from Σ or recursively defined from other regular expressions R_1 and R_2:

1. $R_1 \cdot R_2$ (concatenation),
2. $R_1|R_2$ (union),
3. R_1^* (Kleene star).

Ex: $b(b|ab)^*ab$

✓ *bbbbbabab*
✓ *bbbabbbbab*
✗ *bbbaabbbab*
✗ *baba*
✗ *abab*

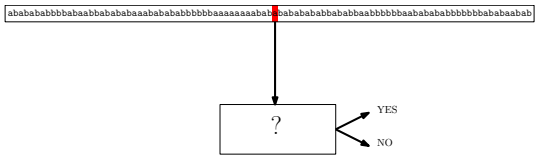
Used in databases, data mining, secret detection, computer networks, protein search...

Given a regular expression R and a string T, **membership** is: check if R matches T.

We define the length of R to be the number of $\cdot$, $|$, and $*$ that it contains.

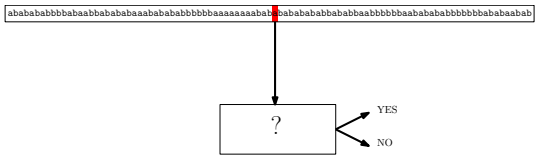
Streaming algorithms

The algorithm first receives and preprocesses the expression.
Next, it keeps reading characters from a very very long string and:



Streaming algorithms

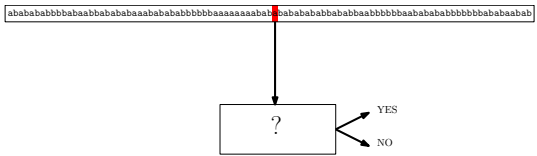
The algorithm first receives and preprocesses the expression.
Next, it keeps reading characters from a very very long string and:



1. **No delay:** After having seen the i-th character, immediately report whether the string seen so far matches the regular expression.

Streaming algorithms

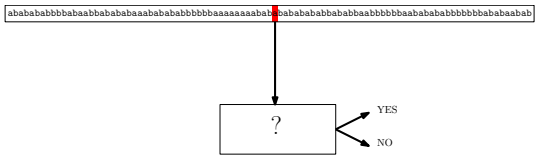
The algorithm first receives and preprocesses the expression.
Next, it keeps reading characters from a very very long string and:



1. **No delay:** After having seen the i-th character, immediately report whether the string seen so far matches the regular expression.
2. **No going back:** Not possible to read any of the earlier characters.

Streaming algorithms

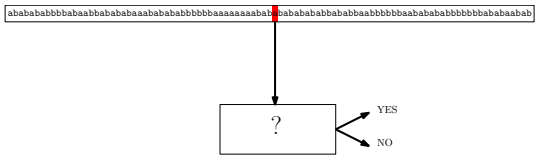
The algorithm first receives and preprocesses the expression.
Next, it keeps reading characters from a very very long string and:



1. **No delay:** After having seen the i-th character, immediately report whether the string seen so far matches the regular expression.
2. **No going back:** Not possible to read any of the earlier characters.
3. **Every space counts:** No access to the original expression (unless stored explicitly).

Streaming algorithms

The algorithm first receives and preprocesses the expression.
Next, it keeps reading characters from a very very long string and:

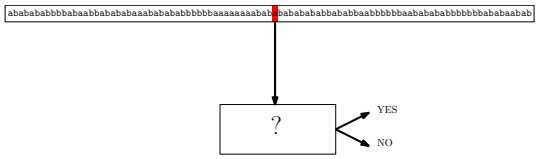


1. **No delay:** After having seen the i-th character, immediately report whether the string seen so far matches the regular expression.
2. **No going back:** Not possible to read any of the earlier characters.
3. **Every space counts:** No access to the original expression (unless stored explicitly).

A crucial tool is the variant of Karp–Rabin fingerprints of [Porat and Porat, FOCS'09].

Streaming algorithms

The algorithm first receives and preprocesses the expression.
Next, it keeps reading characters from a very very long string and:



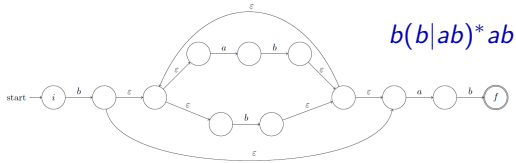
1. **No delay:** After having seen the i-th character, immediately report whether the string seen so far matches the regular expression.
2. **No going back:** Not possible to read any of the earlier characters.
3. **Every space counts:** No access to the original expression (unless stored explicitly).

A crucial tool is the variant of Karp–Rabin fingerprints of [Porat and Porat, FOCS'09].

Classic pattern matching in streaming [Breslauer and Galil, TALG'14] For a pattern of length m, it takes $\mathcal{O}(\log m)$ space and $\mathcal{O}(1)$ time per position.

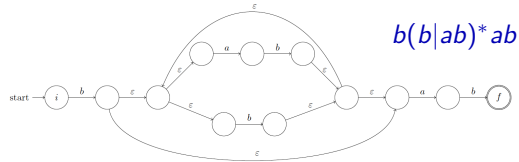
State of the art on regular expression membership

For a regexp of length m,
Classic: Recursively build the **Thompson automaton**, then check if T is accepted.
$\Theta(m)$ space and time/character.



State of the art on regular expression membership

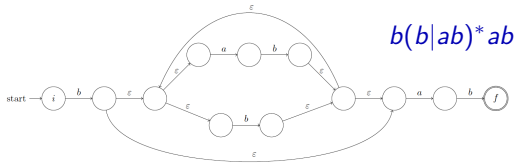
For a regexp of length m,
Classic: Recursively build the **Thompson automaton**, then check if T is accepted.
$\Theta(m)$ space and time/character.



- The best improvements on the time complexity only reduce by logarithmic factors.

State of the art on regular expression membership

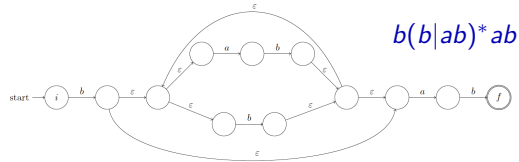
For a regexp of length m,
Classic: Recursively build the **Thompson automaton**, then check if T is accepted.
$\Theta(m)$ space and time/character.



- The best improvements on the time complexity only reduce by logarithmic factors.
- Fine-grained complexity proved conditional lowerbounds, with "hard to match" expressions where improvements are unlikely.

State of the art on regular expression membership

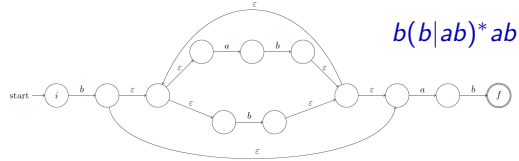
For a regexp of length m,
Classic: Recursively build the **Thompson automaton**, then check if T is accepted.
$\Theta(m)$ space and time/character.



- The best improvements on the time complexity only reduce by logarithmic factors.
- Fine-grained complexity proved conditional lowerbounds, with "hard to match" expressions where improvements are unlikely.
- [Bille and Thorup, SODA'10] $\mathcal{O}(\frac{d \log w}{w} + \log d)$ time/character and $\mathcal{O}(m)$ space, where d is the number of occurrences of $|$ and $*$, and w is the word size.

State of the art on regular expression membership

For a regexp of length m,
Classic: Recursively build the **Thompson automaton**, then check if T is accepted.
$\Theta(m)$ space and time/character.



- The best improvements on the time complexity only reduce by logarithmic factors.
- Fine-grained complexity proved conditional lowerbounds, with "hard to match" expressions where improvements are unlikely.
- [Bille and Thorup, SODA'10] $\mathcal{O}(\frac{d \log w}{w} + \log d)$ time/character and $\mathcal{O}(m)$ space, where d is the number of occurrences of $|$ and $*$, and w is the word size.

What about **space efficiency**?

Space-efficiency for regular expression

For a regexp of length m, $\Theta(m)$ space for the Thompson automaton, can we do better?

Space-efficiency for regular expression

For a regexp of length m, $\Theta(m)$ space for the Thompson automaton, can we do better?

Some intuition: special cases of regexp already studied in streaming

Space-efficiency for regular expression

For a regexp of length m, $\Theta(m)$ space for the Thompson automaton, can we do better?

Some intuition: special cases of regexp already studied in streaming

Problem	Translation to regular expressions	Space complexity

Space-efficiency for regular expression

For a regexp of length m, $\Theta(m)$ space for the Thompson automaton, can we do better?

Some intuition: special cases of regexp already studied in streaming

Problem	Translation to regular expressions	Space complexity
Dictionary matching	$(P_1 \mid \ldots \mid P_d)$	$\mathcal{O}(d \log m)$ [Golan and Porat ESA'17]

Space-efficiency for regular expression

For a regexp of length m, $\Theta(m)$ space for the Thompson automaton, can we do better?

Some intuition: special cases of regexp already studied in streaming

Problem	Translation to regular expressions	Space complexity
Dictionary matching	$(P_1 \mid \ldots \mid P_d)$	$\mathcal{O}(d \log m)$ [Golan and Porat ESA'17]
Wildcards matching	$P_1(1 \mid \ldots \mid \sigma)P_2 \ldots P_d(1 \mid \ldots \mid \sigma)P_{d+1}$	$\mathcal{O}(d \log m)$ [Golan et al., Algorithmica'19]

Space-efficiency for regular expression

For a regexp of length m, $\Theta(m)$ space for the Thompson automaton, can we do better?

Some intuition: special cases of regexp already studied in streaming

Problem	Translation to regular expressions	Space complexity
Dictionary matching	$(P_1\|\dots\|P_d)$	$\mathcal{O}(d \log m)$ [Golan and Porat ESA'17]
Wildcards matching	$P_1(1\|\dots\|\sigma)P_2\dots P_d(1\|\dots\|\sigma)P_{d+1}$	$\mathcal{O}(d \log m)$ [Golan et al., Algorithmica'19]

Dudek, Gawrychowski, Gourdel, Starikovskaya - SODA'22

For any regular expression R with d occurrences of $|$ and $*$, we can solve regular expression membership on a string of length n using $\mathcal{O}(d^3 \operatorname{polylog} n)$ space and $\mathcal{O}(nd^5 \operatorname{polylog} n)$ time per character.

Summary - Streaming Regular Expressions

Complex matching: regular expression, one of the most powerful and versatile model.

Sketch as we go: streaming Karp-Rabin fingerprints as a space efficient way to keep information on the stream.

Dudek, Gawrychowski, Gourdel, Starikovskaya, SODA'22

For any regular expression R with d occurrences of $|$ and $*$, we can solve regular expression membership on a string of length n using $\mathcal{O}(d^3 \operatorname{polylog} n)$ space and $\mathcal{O}(nd^5 \operatorname{polylog} n)$ time per character.

Not shown about this work: technical part, witness mechanism, circuit machinery.

Open questions: is it possible to improve the time complexity while still maintaining space $\sim \operatorname{poly}(d, \log n)$ as well as improving space complexity?

2nd Contribution

Pattern Matching Under Dynamic Time Warping Distance

SPIRE'22



Anne Driemel, Pierre Peterlongo, Tatiana Starikovskaya

Dynamic time warping (DTW) distance: comparing time series

Definition: Minimal distance obtained by duplicating some items (maintaining equal length) and then summing the distances between items at the same position.

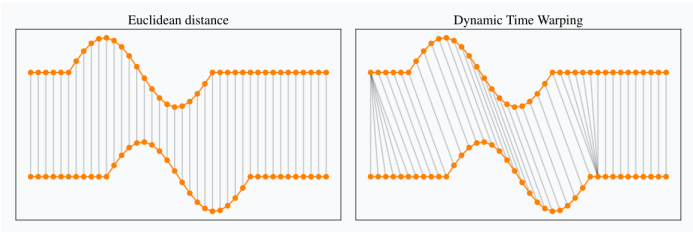


Figure credit: Romain Tavenard

Dynamic time warping (DTW) distance: comparing time series

Definition: Minimal distance obtained by duplicating some items (maintaining equal length) and then summing the distances between items at the same position.

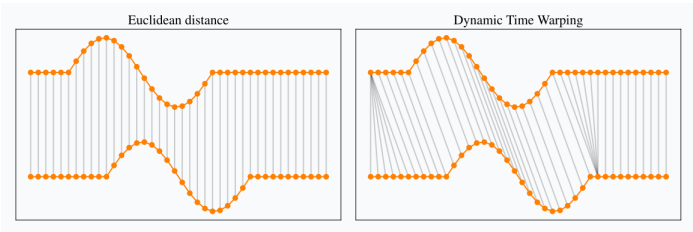


Figure credit: Romain Tavenard

Used in speech recognition to deal with varying speeds and more generally for parametrized curves where each item is a multidimensional point.

Dynamic time warping (DTW) distance: comparing time series

Definition: Minimal distance obtained by duplicating some items (maintaining equal length) and then summing the distances between items at the same position.

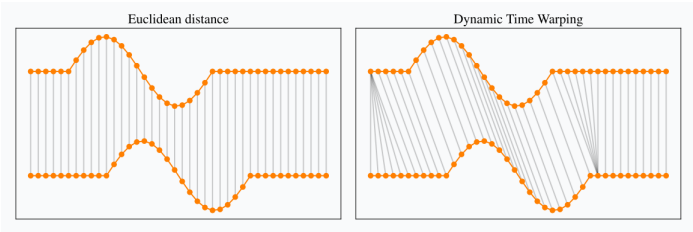


Figure credit: Romain Tavenard

Used in speech recognition to deal with varying speeds and more generally for parametrized curves where each item is a multidimensional point.

But what about **strings**, where the items are characters from a finite alphabet Σ ?

Dynamic time warping (DTW) distance for strings

Given X and Y strings, $\mathrm{DTW}(X, Y)$ is the minimal distance obtained by duplicating some characters (maintaining equal length) and then summing the distances between characters at the same position.

Dynamic time warping (DTW) distance for strings

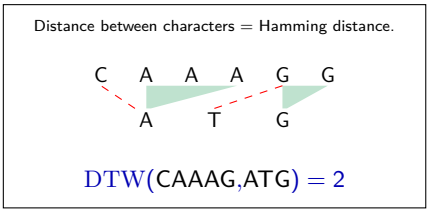
Given X and Y strings, $\mathrm{DTW}(X, Y)$ is the minimal distance obtained by duplicating some characters (maintaining equal length) and then summing the distances between characters at the same position.

Distance between characters = Hamming distance.

$\mathrm{DTW}(\mathrm{CAAAG,ATG}) = 2$

Dynamic time warping (DTW) distance for strings

Given X and Y strings, $\mathrm{DTW}(X, Y)$ is the minimal distance obtained by duplicating some characters (maintaining equal length) and then summing the distances between characters at the same position.



Dynamic Programming

D a matrix of size $(|X|+1)(|Y|+1)$ such that $D[i,j] = \mathrm{DTW}(X[0..i], Y[0..j])$

Initialization $D[0,0] = 0$ and for all (i,j), $D[0,j] = D[i,0] = +\infty$.

Recurrence $D[i,j] = \min\{ \underbrace{D[i-1,j-1]}_{\text{align}}, \underbrace{D[i-1,j]}_{\text{dupl. from Y}}, \underbrace{D[i,j-1]}_{\text{dupl. from X}} \} + d(X[i], Y[j])$.

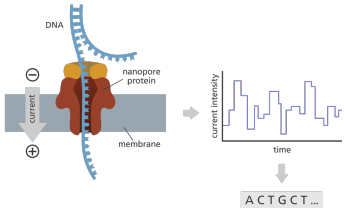
Why is DTW on strings interesting? Third generation sequencing

Why is DTW on strings interesting? Third generation sequencing

Third generation sequencing

A new technique with multiple advantages, but it tends to have errors on the length of homopolymers (/"runs": when the same nucleotide is repeated)...

Figure credit: yourgenome.org

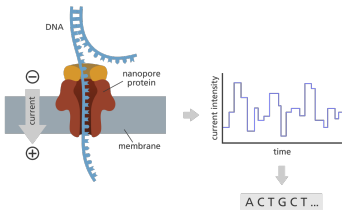


Why is DTW on strings interesting? Third generation sequencing

Third generation sequencing

A new technique with multiple advantages, but it tends to have errors on the length of homopolymers (/"runs": when the same nucleotide is repeated)...

Figure credit: yourgenome.org



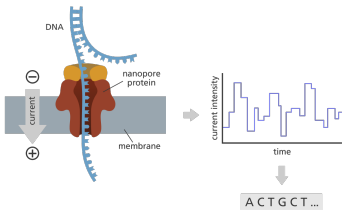
DTW was already used to align the electric signal by [Loose et al. 2016] and [Han et al. 2020].

Why is DTW on strings interesting? Third generation sequencing

Third generation sequencing

A new technique with multiple advantages, but it tends to have errors on the length of homopolymers (/"runs": when the same nucleotide is repeated)...

Figure credit: yourgenome.org



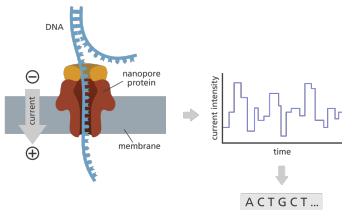
DTW was already used to align the electric signal by [Loose et al. 2016] and [Han et al. 2020]. We propose to use it to align the reads (strings) to a reference (string).

Why is DTW on strings interesting? Third generation sequencing

Third generation sequencing

A new technique with multiple advantages, but it tends to have errors on the length of homopolymers (/"runs": when the same nucleotide is repeated)...

Figure credit: yourgenome.org



DTW was already used to align the electric signal by [Loose et al. 2016] and [Han et al. 2020].
We propose to use it to align the reads (strings) to a reference (string).
So we want to find the positions where a string aligns best ($\neq$ similarity btw. two strings)?

Pattern matching DTW

Input two strings P and T.
Output for every $0 \leq j < |T|$, $\min_{0 \leq i \leq j} \mathrm{DTW}(P, T[i..j])$.

Dynamic Programming

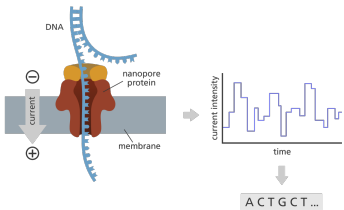
Changes the Initialization
for all $0 \leq j \leq |T|$, $D[0, j] = 0$ and
for all $1 \leq i \leq |P|$, $D[i, 0] = +\infty$.

Why is DTW on strings interesting? Third generation sequencing

Third generation sequencing

A new technique with multiple advantages, but it tends to have errors on the length of homopolymers (/"runs": when the same nucleotide is repeated)...

Figure credit: yourgenome.org



DTW was already used to align the electric signal by [Loose et al. 2016] and [Han et al. 2020].
We propose to use it to align the reads (strings) to a reference (string).
So we want to find the positions where a string aligns best ($\neq$ similarity btw. two strings)?

Pattern matching DTW

Input two strings P and T.
Output for every $0 \leq j < |T|$, $\min_{0 \leq i \leq j} \mathrm{DTW}(P, T[i..j])$.

Dynamic Programming

Changes the Initialization
for all $0 \leq j \leq |T|$, $D[0, j] = 0$ and
for all $1 \leq i \leq |P|$, $D[i, 0] = +\infty$.

State of the art for DTW on strings

X and Y strings, $N = |X|$ and $M = |Y|$, compute $\mathrm{DTW}(X, Y)$.

State of the art for DTW on strings

X and Y strings, $N = |X|$ and $M = |Y|$, compute $\mathrm{DTW}(X, Y)$.

The dynamic programming takes $\mathcal{O}(NM)$ time $= \mathcal{O}(N^2)$ in the case $N = M$.

State of the art for DTW on strings

X and Y strings, $N = |X|$ and $M = |Y|$, compute $\mathrm{DTW}(X, Y)$.

The dynamic programming takes $\mathcal{O}(NM)$ time $= \mathcal{O}(N^2)$ in the case $N = M$.

A conditional lowerbound by [Abboud et al. FOCS'15] indicates that strongly subquadratic algorithms are unlikely.

State of the art for DTW on strings

X and Y strings, $N = |X|$ and $M = |Y|$, compute $\mathrm{DTW}(X, Y)$.

The dynamic programming takes $\mathcal{O}(NM)$ time $= \mathcal{O}(N^2)$ in the case $N = M$.

A conditional lowerbound by [Abboud et al. FOCS'15] indicates that strongly subquadratic algorithms are unlikely.

What if X and Y are made of consecutive repetitions of the same character "runs", n and m runs respectively?

State of the art for DTW on strings

X and Y strings, $N = |X|$ and $M = |Y|$, compute $\mathrm{DTW}(X, Y)$.

The dynamic programming takes $\mathcal{O}(NM)$ time $= \mathcal{O}(N^2)$ in the case $N = M$.

A conditional lowerbound by [Abboud et al. FOCS'15] indicates that strongly subquadratic algorithms are unlikely.

What if X and Y are made of consecutive repetitions of the same character "runs", n and m runs respectively?

Run-length compressed
$\mathrm{DTW}(X, Y)$ can be computed $\mathcal{O}(mN + Mn)$-time [Froese et al. Algorithmica'22]

State of the art for DTW on strings

X and Y strings, $N = |X|$ and $M = |Y|$, compute $\mathrm{DTW}(X, Y)$.

The dynamic programming takes $\mathcal{O}(NM)$ time $= \mathcal{O}(N^2)$ in the case $N = M$.

A conditional lowerbound by [Abboud et al. FOCS'15] indicates that strongly subquadratic algorithms are unlikely.

What if X and Y are made of consecutive repetitions of the same character "runs", n and m runs respectively?

Run-length compressed
$\mathrm{DTW}(X, Y)$ can be computed $\mathcal{O}(mN + Mn)$-time [Froese et al. Algorithmica'22]

Low distance regime

State of the art for DTW on strings

X and Y strings, $N = |X|$ and $M = |Y|$, compute $\mathrm{DTW}(X, Y)$.

The dynamic programming takes $\mathcal{O}(NM)$ time $= \mathcal{O}(N^2)$ in the case $N = M$.

A conditional lowerbound by [Abboud et al. FOCS'15] indicates that strongly subquadratic algorithms are unlikely.

What if X and Y are made of consecutive repetitions of the same character "runs", n and m runs respectively?

Run-length compressed

$\mathrm{DTW}(X, Y)$ can be computed $\mathcal{O}(mN + Mn)$-time [Froese et al. Algorithmica'22]

Low distance regime

If $\mathrm{DTW}(X, Y) \leq k$ it can be computed in $\mathcal{O}(kN)$-time [Kuszmaul ICALP'19]

State of the art for DTW on strings

X and Y strings, $N = |X|$ and $M = |Y|$, compute $\mathrm{DTW}(X, Y)$.

The dynamic programming takes $\mathcal{O}(NM)$ time $= \mathcal{O}(N^2)$ in the case $N = M$.

A conditional lowerbound by [Abboud et al. FOCS'15] indicates that strongly subquadratic algorithms are unlikely.

What if X and Y are made of consecutive repetitions of the same character "runs", n and m runs respectively?

Run-length compressed

$\mathrm{DTW}(X, Y)$ can be computed $\mathcal{O}(mN + Mn)$-time [Froese et al. Algorithmica'22]

Low distance regime

If $\mathrm{DTW}(X, Y) \leq k$ it can be computed in $\mathcal{O}(kN)$-time [Kuszmaul ICALP'19]

Can we also use (and combine) those approaches for Pattern Matching?

Runs simplify the dynamic programming matrix

T and P strings, $N = |T|$ and $M = |P|$, with n and m runs.

Runs simplify the dynamic programming matrix

T and P strings, $N = |T|$ and $M = |P|$, with n and m runs.

Runs simplify the dynamic programming matrix

T and P strings, $N = |T|$ and $M = |P|$, with n and m runs.

		G	G	T	T	T	T	C	T	T	A	T	T	T	T	G	G	T	G	A	T	A
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	∞	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0
A	∞	2	2	2	2	2	2	2	2	2	0	1	2	2	2	2	2	2	2	0	1	0
T	∞	3	3	2	2	2	2	3	2	2	1	0	0	0	0	1	2	2	3	1	0	1
T	∞	4	4	2	2	2	2	3	2	2	2	0	0	0	0	1	2	2	3	2	0	1
A	∞	5	5	3	3	3	3	3	3	3	2	1	1	1	1	1	2	3	3	2	1	0
T	∞	6	6	3	3	3	3	4	3	3	3	1	1	1	1	2	2	2	3	3	1	1

Runs simplify the dynamic programming matrix

T and P strings, $N = |T|$ and $M = |P|$, with n and m runs.

		G	G	T	T	T	T	C	T	T	A	T	T	T	T	G	G	T	G	A	T	A
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	∞	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0
A	∞	2	2	2	2	2	2	2	2	2	0	1	2	2	2	2	2	2	2	0	1	0
T	∞	3	3	2	2	2	2	3	2	2	1	0	0	0	0	1	2	2	3	1	0	1
T	∞	4	4	2	2	2	2	3	2	2	2	0	0	0	0	1	2	2	3	2	0	1
A	∞	5	5	3	3	3	3	3	3	3	2	1	1	1	1	1	2	3	3	2	1	0
T	∞	6	6	3	3	3	3	4	3	3	3	1	1	1	1	2	2	2	3	3	1	1

If $P[i_p..j_p]$ is a run in P, and $T[i_t..j_t]$ is a run in T, we call $D[i_p..j_p, i_t..j_t]$ a ***block***:

Runs simplify the dynamic programming matrix

T and P strings, $N = |T|$ and $M = |P|$, with n and m runs.

		G	G	T	T	T	T	C	T	T	A	T	T	T	T	G	G	T	G	A	T	A
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	∞	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0
A	∞	2	2	2	2	2	2	2	2	2	0	1	2	2	2	2	2	2	2	0	1	0
T	∞	3	3	2	2	2	2	3	2	2	1	0	0	0	0	1	2	2	3	1	0	1
T	∞	4	4	2	2	2	2	3	2	2	2	0	0	0	0	1	2	2	3	2	0	1
A	∞	5	5	3	3	3	3	3	3	3	2	1	1	1	1	1	2	3	3	2	1	0
T	∞	6	6	3	3	3	3	4	3	3	3	1	1	1	1	2	2	2	3	3	1	1

If $P[i_p..j_p]$ is a run in P, and $T[i_t..j_t]$ is a run in T, we call $D[i_p..j_p, i_t..j_t]$ a ***block***:

$$D[i,j] = \min\{D[i-1,j-1], D[i-1,j], D[i,j-1]\} + \underbrace{d(T[i], P[j])}_{\text{Constant!}}$$

Runs simplify the dynamic programming matrix

T and P strings, $N = |T|$ and $M = |P|$, with n and m runs.

		G	G	T	T	T	T	C	T	T	A	T	T	T	T	G	G	T	G	A	T	A
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	∞	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0
A	∞	2	2	2	2	2	2	2	2	2	0	1	2	2	2	2	2	2	2	0	1	0
T	∞	3	3	2	2	2	2	3	2	2	1	0	0	0	0	1	2	2	3	1	0	1
T	∞	4	4	2	2	2	2	3	2	2	2	0	0	0	0	1	2	2	3	2	0	1
A	∞	5	5	3	3	3	3	3	3	3	2	1	1	1	1	1	2	3	3	2	1	0
T	∞	6	6	3	3	3	3	4	3	3	3	1	1	1	1	2	2	2	3	3	1	1

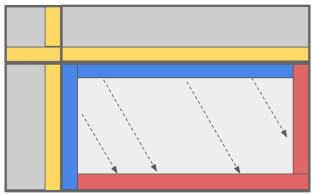
If $P[i_p..j_p]$ is a run in P, and $T[i_t..j_t]$ is a run in T, we call $D[i_p..j_p, i_t..j_t]$ a ***block***:

$$D[i,j] = \min\{D[i-1,j-1], D[i-1,j], D[i,j-1]\} + \underbrace{d(T[i], P[j])}_{\text{Constant!}}$$

Inside a block we can take the shortest path

Runs simplify the dynamic programming matrix

T and P strings, $N = |T|$ and $M = |P|$, with n and m runs.



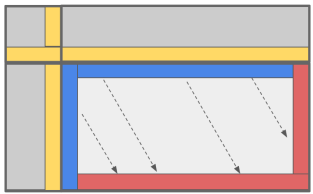
If $P[i_p..j_p]$ is a run in P, and $T[i_t..j_t]$ is a run in T, we call $D[i_p..j_p, i_t..j_t]$ a ***block***:

$$D[i,j] = \min\{D[i-1,j-1], D[i-1,j], D[i,j-1]\} + \underbrace{d(T[i], P[j])}_{\text{Constant!}}$$

Inside a block we can take the shortest path

Runs simplify the dynamic programming matrix

T and P strings, $N = |T|$ and $M = |P|$, with n and m runs.



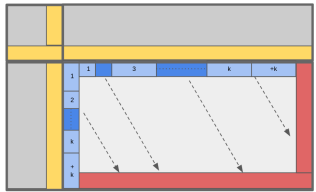
If $P[i_p..j_p]$ is a run in P, and $T[i_t..j_t]$ is a run in T, we call $D[i_p..j_p, i_t..j_t]$ a ***block***:

$$D[i,j] = \min\{D[i-1,j-1], D[i-1,j], D[i,j-1]\} + \underbrace{d(T[i], P[j])}_{\text{Constant!}}$$

Inside a block we can take the shortest path $\rightarrow \mathcal{O}(Nm + Mn)$ time.

Runs simplify the dynamic programming matrix

T and P strings, $N = |T|$ and $M = |P|$, with n and m runs.



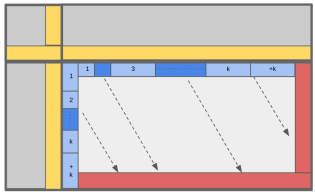
If $P[i_p..j_p]$ is a run in P, and $T[i_t..j_t]$ is a run in T, we call $D[i_p..j_p, i_t..j_t]$ a ***block***:

$$D[i,j] = \min\{D[i-1,j-1], D[i-1,j], D[i,j-1]\} + \underbrace{d(T[i], P[j])}_{\text{Constant!}}$$

Inside a block we can take the shortest path $\rightarrow \mathcal{O}(Nm + Mn)$ time.
If d is an integer distance, the values $\leq k$ can take at most k distinct values...

Runs simplify the dynamic programming matrix

T and P strings, $N = |T|$ and $M = |P|$, with n and m runs.



If $P[i_p..j_p]$ is a run in P, and $T[i_t..j_t]$ is a run in T, we call $D[i_p..j_p, i_t..j_t]$ a ***block***:

$$D[i,j] = \min\{D[i-1,j-1], D[i-1,j], D[i,j-1]\} + \underbrace{d(T[i], P[j])}_{\text{Constant!}}$$

Inside a block we can take the shortest path $\rightarrow \mathcal{O}(Nm + Mn)$ time.
If d is an integer distance, the values $\leq k$ can take at most k distinct values...

Driemel, Gourdel, Peterlongo, Starikovskaya - SPIRE'22

For an integer distance $d : \Sigma \times \Sigma \rightarrow \mathbb{Z}^+$, given P and T run-length compressed with m and n runs resp., we can compute all DTW distances $\leq k$ in $\mathcal{O}(kmn)$ time.

Summary - Pattern Matching for DTW

Similarity measure: the Dynamic Time Warping distance on string can be used to align reads with homopolymers errors onto a reference genome.

Sketch as input: Run-length compression sketch, suited to the metric, that allows a complexity proportional to the size of the sketch not the original input.

Driemel, Gourdel, Peterlongo, Starikovskaya - SPIRE'22

For an integer distance $d : \Sigma \times \Sigma \to \mathbb{Z}^+$, given P and T run-length compressed with m and n runs resp., we can compute all DTW distances $\leq k$ in $\mathcal{O}(kmn)$ time.

Not shown about this work: $\mathcal{O}(n+m)$ algorithm for $k=1$, approximations results, and experiments.

Open questions: faster algorithms? $\tilde{\mathcal{O}}(nm)$ [Boneh et al. arxiv'23]
Is $\tilde{\mathcal{O}}(k(n+m))$ space and time possible?

3rd Contribution

Optimal Square Detection Over General Alphabets

SODA'23


Jonas Ellert, Paweł Gawrychowski

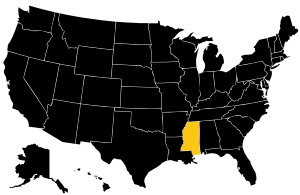
Squares and repetition detection

For a string T, $T[i..j]$ is a **square** if there exists a string x such that $T[i..j] = xx$.

Squares and repetition detection

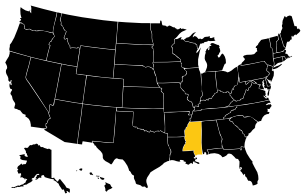
For a string T, $T[i..j]$ is a **square** if there exists a string x such that $T[i..j] = xx$.

m i s s i s s i p p i



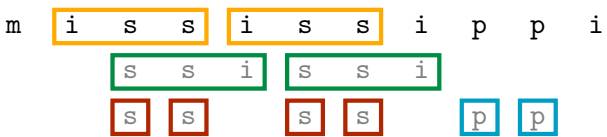
Squares and repetition detection

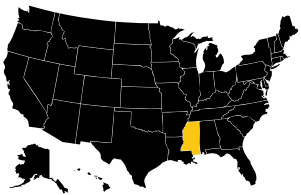
For a string T, $T[i..j]$ is a **square** if there exists a string x such that $T[i..j] = xx$.



Squares and repetition detection

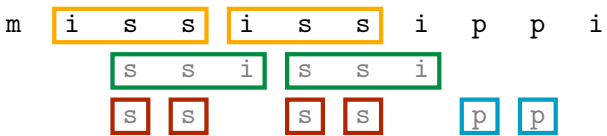
For a string T, $T[i..j]$ is a **square** if there exists a string x such that $T[i..j] = xx$.

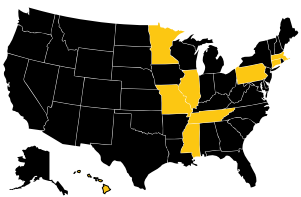




Squares and repetition detection

For a string T, $T[i..j]$ is a **square** if there exists a string x such that $T[i..j] = xx$.

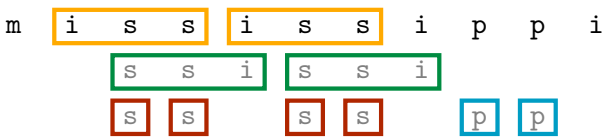


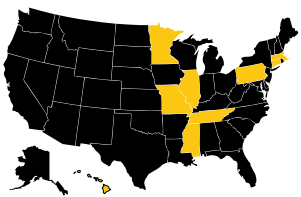


Connecticut, Hawaii, Illinois, Massachusetts, Minnesota.

Squares and repetition detection

For a string T, $T[i..j]$ is a **square** if there exists a string x such that $T[i..j] = xx$.





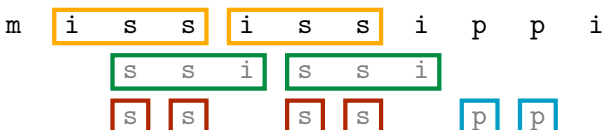
Connecticut, Hawaii, Illinois, Massachusetts, Minnesota.

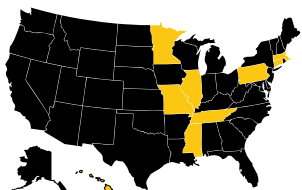
Some problems regarding squares:

- test square-freeness, i.e., whether or not a string contains a square

Squares and repetition detection

For a string T, $T[i..j]$ is a **square** if there exists a string x such that $T[i..j] = xx$.





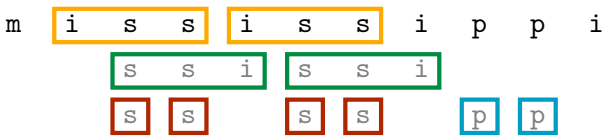
Connecticut, Hawaii, Illinois, Massachusetts, Minnesota.

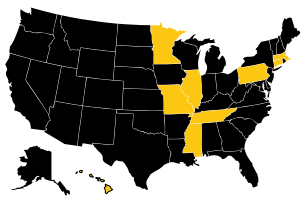
Some problems regarding squares:

- test square-freeness, i.e., whether or not a string contains a square
- compute all squares (or runs) in a given string

Squares and repetition detection

For a string T, $T[i..j]$ is a **square** if there exists a string x such that $T[i..j] = xx$.





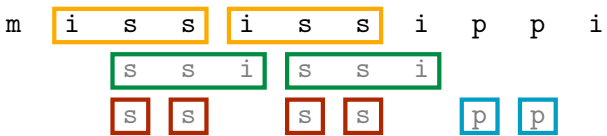
Connecticut, Hawaii, Illinois, Massachusetts, Minnesota.

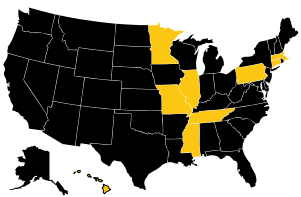
Some problems regarding squares:

- test square-freeness, i.e., whether or not a string contains a square
- compute all squares (or runs) in a given string
- compute all distinct squares in a given string

Squares and repetition detection

For a string T, $T[i..j]$ is a **square** if there exists a string x such that $T[i..j] = xx$.





Connecticut, Hawaii, Illinois, Massachusetts, Minnesota.

Some problems regarding squares:

- **test square-freeness, i.e., whether or not a string contains a square**
- compute all squares (or runs) in a given string
- compute all distinct squares in a given string

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time
  e.g. comparison sort

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time
  e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time — e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time — e.g. KMP pattern matching

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time — e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time — e.g. KMP pattern matching

	LSA	GOA	GUA
Lempel-Ziv Suffix Sorting			

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time
  e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time
  e.g. KMP pattern matching

	LSA	GOA	GUA
Lempel-Ziv	$\mathcal{O}(n)$		
Suffix Sorting	$\mathcal{O}(n)$		

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time — e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time — e.g. KMP pattern matching

	LSA	GOA	GUA
Lempel-Ziv	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	
Suffix Sorting	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time — e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time — e.g. KMP pattern matching

	LSA	GOA	GUA
Lempel-Ziv	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Suffix Sorting	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time — e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time — e.g. KMP pattern matching

	LSA	GOA	GUA
Lempel-Ziv	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Suffix Sorting	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Square-Freeness	$\boldsymbol{\mathcal{O}(n)}$		
	Kolpakov & Kucherov '99		

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time — e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time — e.g. KMP pattern matching

	LSA	GOA	GUA
Lempel-Ziv	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Suffix Sorting	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Square-Freeness	$\boldsymbol{\mathcal{O}(n)}$	$\boldsymbol{\Theta(n)}$	
	Kolpakov & Kucherov '99	**Ellert & Fischer '21**	

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time — e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time — e.g. KMP pattern matching

	LSA	GOA	GUA
Lempel-Ziv	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Suffix Sorting	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Square-Freeness	$\boldsymbol{\mathcal{O}(n)}$	$\boldsymbol{\Theta(n)}$	$\boldsymbol{\mathcal{O}(n \lg n)}$
	Kolpakov & Kucherov '99	**Ellert & Fischer '21**	**e.g. Main & Lorentz '84**

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time — e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time — e.g. KMP pattern matching

	LSA	GOA	GUA
Lempel-Ziv	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Suffix Sorting	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Square-Freeness	$\boldsymbol{\mathcal{O}(n)}$	$\boldsymbol{\Theta(n)}$	$\boldsymbol{\mathcal{O}(n \lg n)}$
	Kolpakov & Kucherov '99	**Ellert & Fischer '21**	**e.g. Main & Lorentz '84**

optimal for $\sigma = n$

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time — e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time — e.g. KMP pattern matching

	LSA	GOA	GUA
Lempel-Ziv	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Suffix Sorting	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Square-Freeness	$\boldsymbol{\mathcal{O}(n)}$	$\boldsymbol{\Theta(n)}$	$\boldsymbol{\mathcal{O}(n \lg n)}$
	Kolpakov & Kucherov '99	**Ellert & Fischer '21**	**e.g. Main & Lorentz '84**

optimal for $\sigma = n$
but what about $\sigma < n$?

Alphabet assumptions for T of length n, with $|\Sigma| = \sigma$

- **linearly-sortable alphabet (LSA)**: sort n symbols in $\mathcal{O}(n)$ time — e.g. $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\{0, \ldots, 255\}$, or $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**: test $T[i] < T[j]$ in constant time — e.g. comparison sort
- **general unordered alphabet (GUA)**: test $T[i] = T[j]$ in constant time — e.g. KMP pattern matching

	LSA	GOA	GUA
Lempel-Ziv	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Suffix Sorting	$\mathcal{O}(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Square-Freeness	$\boldsymbol{\mathcal{O}(n)}$	$\boldsymbol{\Theta(n)}$	$\boldsymbol{\mathcal{O}(n \lg n)}$
	Kolpakov & Kucherov '99	**Ellert & Fischer '21**	**e.g. Main & Lorentz '84**

optimal for $\sigma = n$
but what about $\sigma < n$?

$\Theta(n \lg \sigma)$
Our contribution

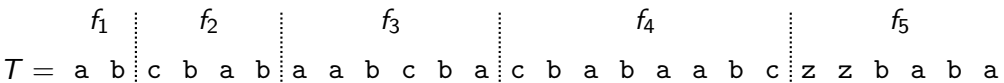
Sketching: Δ-approximate Lempel-Ziv factorisation

Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$

Sketching: Δ-approximate Lempel-Ziv factorisation

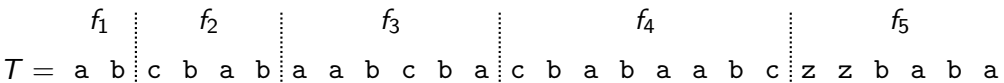
Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$



Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$ such that

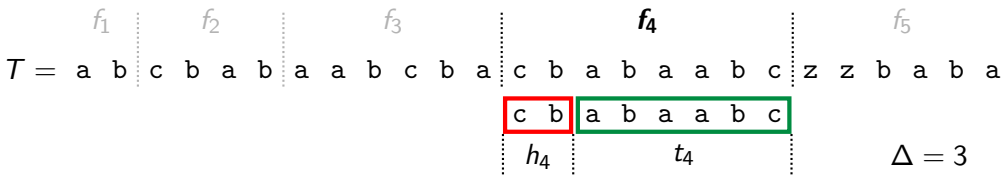
- $|f_i| > 0$ and $f_i = h_i t_i$ (*head* and *tail*) with $|h_i| < \Delta$



Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$ such that

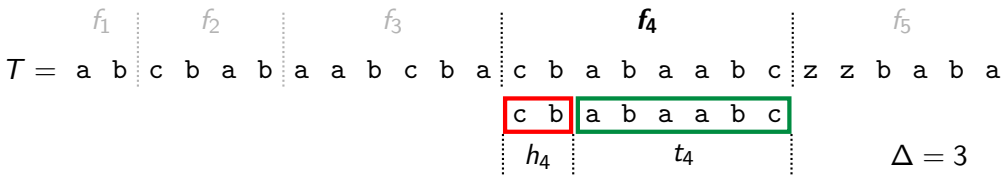
- $|f_i| > 0$ and $f_i = h_i t_i$ (*head* and *tail*) with $|h_i| < \Delta$



Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$ such that

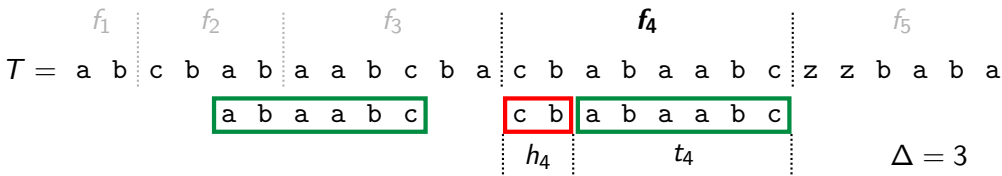
- $|f_i| > 0$ and $f_i = h_i t_i$ (*head* and *tail*) with $|h_i| < \Delta$, and
- t_i is empty or occurs at least twice in $f_1 f_2 \dots f_i$



Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$ such that

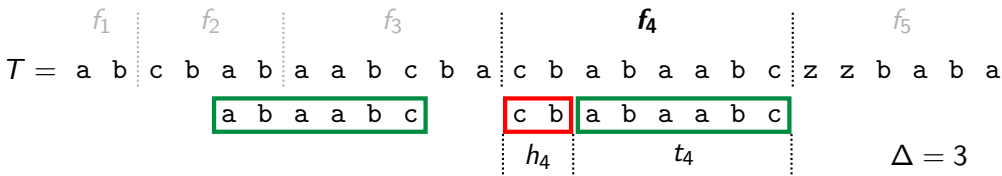
- $|f_i| > 0$ and $f_i = h_i t_i$ (*head* and *tail*) with $|h_i| < \Delta$, and
- t_i is empty or occurs at least twice in $f_1 f_2 \dots f_i$



Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$ such that

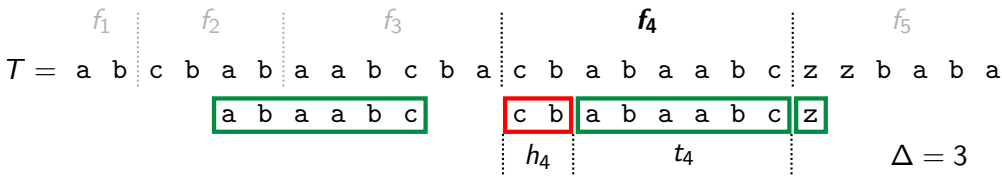
- $|f_i| > 0$ and $f_i = h_i t_i$ (*head* and *tail*) with $|h_i| < \Delta$, and
- t_i is empty or occurs at least twice in $f_1 f_2 \dots f_i$, and
- $f_i f_{i+1}[1]$ does not occur in $f_1 f_2 \dots f_i$ (or $i = z$).



Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$ such that

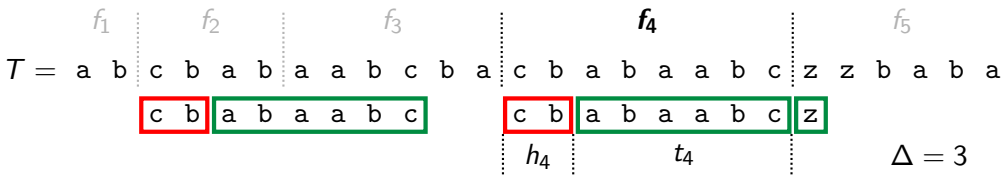
- $|f_i| > 0$ and $f_i = h_i t_i$ (*head* and *tail*) with $|h_i| < \Delta$, and
- t_i is empty or occurs at least twice in $f_1 f_2 \dots f_i$, and
- $f_i f_{i+1}[1]$ does not occur in $f_1 f_2 \dots f_i$ (or $i = z$).



Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$ such that

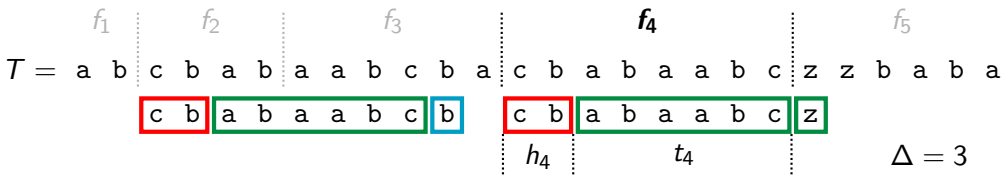
- $|f_i| > 0$ and $f_i = h_i t_i$ (*head* and *tail*) with $|h_i| < \Delta$, and
- t_i is empty or occurs at least twice in $f_1 f_2 \dots f_i$, and
- $f_i f_{i+1}[1]$ does not occur in $f_1 f_2 \dots f_i$ (or $i = z$).



Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$ such that

- $|f_i| > 0$ and $f_i = h_i t_i$ (*head* and *tail*) with $|h_i| < \Delta$, and
- t_i is empty or occurs at least twice in $f_1 f_2 \dots f_i$, and
- $f_i f_{i+1}[1]$ does not occur in $f_1 f_2 \dots f_i$ (or $i = z$).

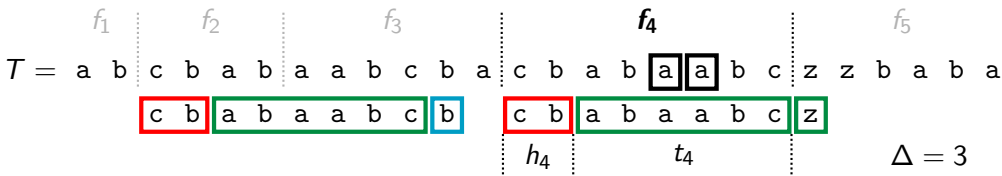


- no need to detect squares that are entirely contained in t_i

Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$ such that

- $|f_i| > 0$ and $f_i = h_i t_i$ (*head* and *tail*) with $|h_i| < \Delta$, and
- t_i is empty or occurs at least twice in $f_1 f_2 \dots f_i$, and
- $f_i f_{i+1}[1]$ does not occur in $f_1 f_2 \dots f_i$ (or $i = z$).

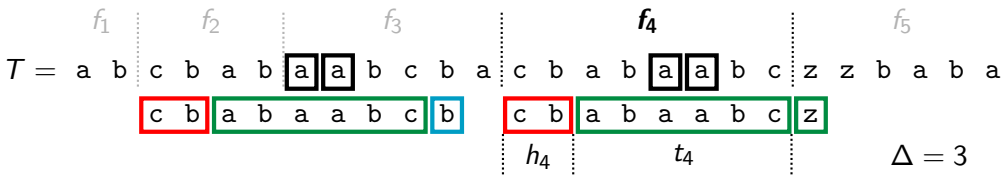


- no need to detect squares that are entirely contained in t_i

Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$ such that

- $|f_i| > 0$ and $f_i = h_i t_i$ (*head* and *tail*) with $|h_i| < \Delta$, and
- t_i is empty or occurs at least twice in $f_1 f_2 \dots f_i$, and
- $f_i f_{i+1}[1]$ does not occur in $f_1 f_2 \dots f_i$ (or $i = z$).

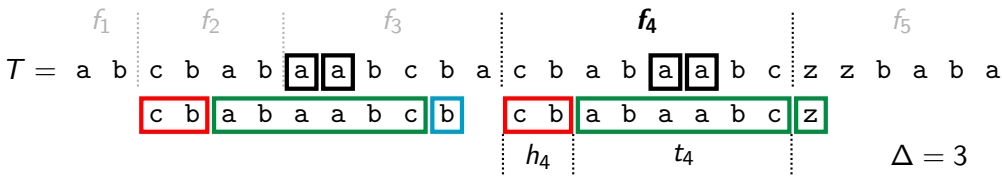


- no need to detect squares that are entirely contained in t_i

Sketching: Δ-approximate Lempel-Ziv factorisation

Δ-approximate Lempel-Ziv factorisation: $T = f_1 f_2 \dots f_z$ such that

- $|f_i| > 0$ and $f_i = h_i t_i$ (*head* and *tail*) with $|h_i| < \Delta$, and
- t_i is empty or occurs at least twice in $f_1 f_2 \dots f_i$, and
- $f_i f_{i+1}[1]$ does not occur in $f_1 f_2 \dots f_i$ (or $i = z$).



- no need to detect squares that are entirely contained in t_i

We use this "easier to compute" approximation to detect squares.

Summary - Squares Over Unordered Alphabets

Repetition detection: We analysed the complexity of square detection in the most general model where they are defined.

Sketch: The Δ-approximate Lempel–Ziv factorization relaxes the constraints on phrases starting position to make it easier to compute.

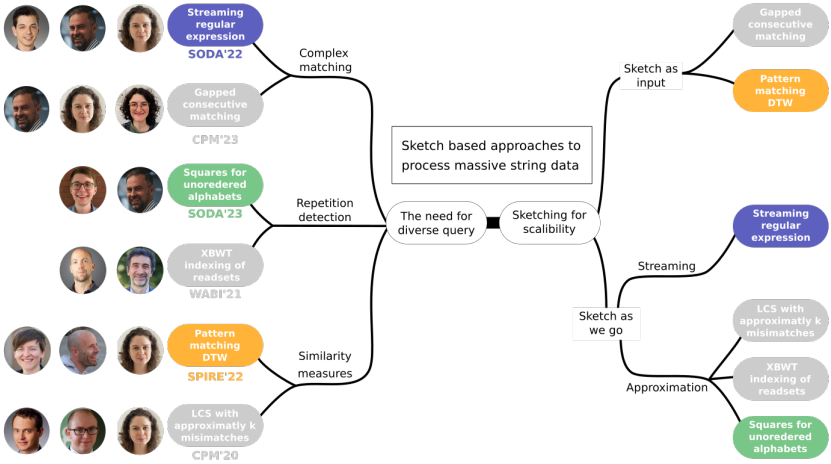
Ellert, Gawrychowski, Gourdel - SODA'23

Testing square-freeness of a length-n string that contains σ distinct symbols from a general unordered alphabet can be done in $\mathcal{O}(n \log \sigma)$

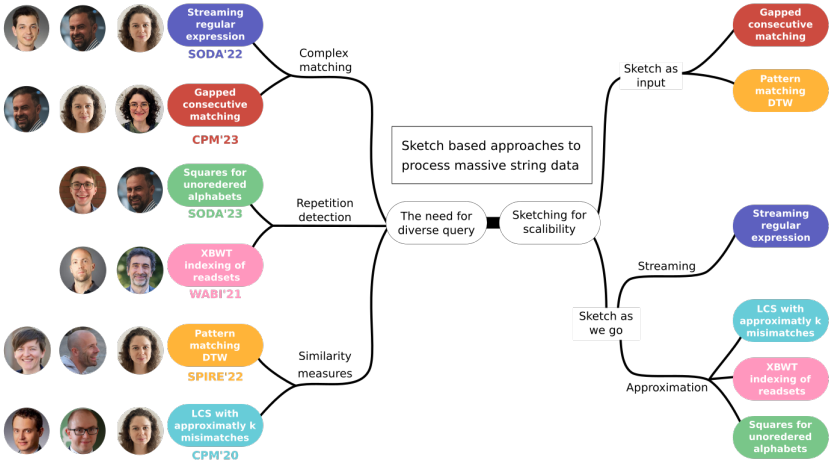
Not shown about this work: how to use the sketch, lower bounds, extension to runs, construction of the factorization...

Open questions: adapt the lower bound to randomized algorithms ?

Conclusion


Sketch based approaches to process massive string data
The need for diverse query
Sketching for scalibility
Complex matching
Repetition detection
Similarity measures
Sketch as input
Sketch as we go
Streaming
Approximation
Streaming regular expression
SODA'22
Gapped consecutive matching
CPM'23
Squares for unoredered alphabets
SODA'23
XBWT indexing of readsets
WABI'21
Pattern matching DTW
SPIRE'22
LCS with approximatly k misimatches
CPM'20
Gapped consecutive matching
Pattern matching DTW
Streaming regular expression
LCS with approximatly k misimatches
XBWT indexing of readsets
Squares for unoredered alphabets

Conclusion


Sketch based approaches to process massive string data
The need for diverse query
Sketching for scalibility
Complex matching
Repetition detection
Similarity measures
Streaming regular expression
SODA'22
Gapped consecutive matching
CPM'23
Squares for unoredered alphabets
SODA'23
XBWT indexing of readsets
WABI'21
Pattern matching DTW
SPIRE'22
LCS with approximatly k misimatches
CPM'20
Sketch as input
Sketch as we go
Streaming
Approximation
Gapped consecutive matching
Pattern matching DTW
Streaming regular expression
LCS with approximatly k misimatches
XBWT indexing of readsets
Squares for unoredered alphabets

Conclusion

Main thesis: Sketches (as input or computed on the fly) help in improving the complexity of most processing tasks.

Future directions:

- Δ-approximate Lempel-Ziv factorization, study the links with prefix-free parsing and evaluate potential practical applications.
- Gapped consecutive matching: explore connection to spaced seeds in Bioinformatics.
- Grammar compression: a lot of recent progress with balanced grammars and local consistency. I would be interested in the construction phase of grammar because it seems to be the main practical downside...

Thank you for your attention!

Appendix

Streaming Regular Expression Membership and Pattern Matching

SODA'22



Bartłomiej Dudek, Paweł Gawrychowski, Tatiana Starikovskaya

Dudek, Gawrychowski, Gourdel, Starikovskaya, SODA'22

For any regular expression R with d occurrences of $|$ and $*$, we can solve regular expression membership on a text of length n using $\mathcal{O}(d^3 \operatorname{polylog} n)$ space and $\mathcal{O}(nd^5 \operatorname{polylog} n)$ time per character.

Main steps:

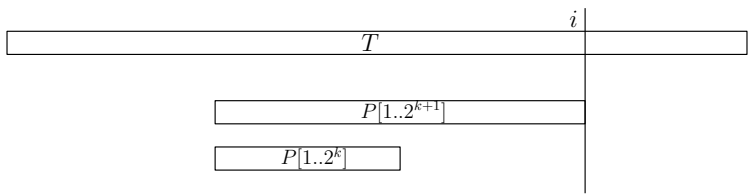
- Define atomic strings: "words" in the regular expression.
- Store efficiently specific occurrences of prefixes of atomic strings.
- Link those occurences stored to test if there is a "partial" match of R.
- Translate this in a graph problem.
- Solve this problem efficiently by a circuit machinery!

We also improve the circuit framework.

Streaming pattern matching

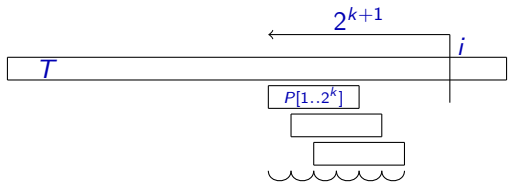
Porat and Porat, FOCS'09 $\mathcal{O}(\log m \cdot \log n)$ bits of space, $\mathcal{O}(\log m)$ time per position.

Main idea: Given an algorithm $\mathcal{A}_k$ which generates all occurrences of $P[1..2^k]$, we will develop a new algorithm $\mathcal{A}_{k+1}$ which generates all occurrences of $P[1..2^{k+1}]$ using just $\mathcal{O}(\log n)$ bits of additional memory.



But... What if we have may occurrences of $P[1..2^k]$ in the window of size 2^{k+1}?

What if we have may occurrences of $P[1..2^k]$ in the window of size 2^{k+1}?



Then there is periodicity! The occurences can be stored efficiently:

- The start, the period, and the number of repetitions
- The fingerprint of the prefix of the text up to the start of the progression and the fingerprint of the period

$\mathcal{O}(\log n)$ bits of space per level!

Correctness: if P occurs, all $\log m$ prefixes will be there too and be detected w.h.p.

Main idea

We run a separate pattern matching process for each of the atomic strings, and store some of the found occurrences as witnesses.

Witness

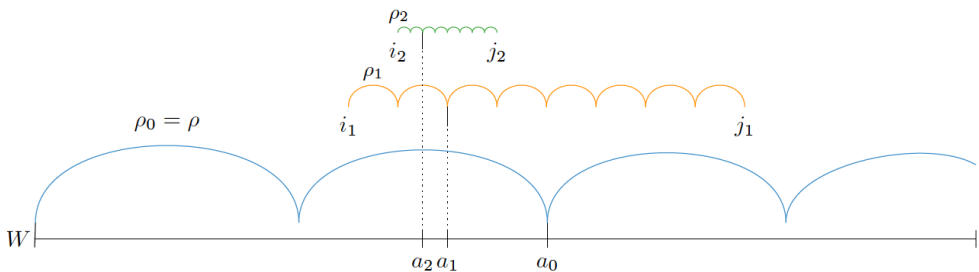
Let P be a prefix of length 2^i of some atomic string that occurs ending at position r in T. Then, r is a witness if $T[1..r]$ is a partial occurrence of R ending at P.

This works if everything is aperiodic:
there are only a constant number of witnesses for each prefix.

However, this is not always the case...

For the general case:

- Periodic atomic string will be found in periodic regions of the text.
- We keep some occurrences of earlier atomic strings on an accepting path (check acceptance via a graph problem).
- We carefully chose them to always be able to bound their number.



Main novelty: Treat the non-periodic and periodic cases together with a reasoning $\mathcal{O}(\log n)$ levels.

Circuit framework

$$C_k[u,v][d] = \bigvee_{\substack{w \in V(G) \\ i \in \{0,\ldots,d\}}} C_{k-1}[u,w][i] \wedge C_{k-1}[w,v][d-i]$$

Lokshtanov and Nederlof, STOC'10; Bringmann, SODA'17

Consider a circuit C with every gate being an addition or convolution gate and computing a vectors of the same length. Assuming that no convolution gate overflows, we can efficiently compute an entry of the output vector in space depending on the size of the circuit (and not the length of the vectors).

We remove the dependency on the Extended Riemann Hypothesis, replacing it with an application of the Bombieri–Vinogradov theorem.

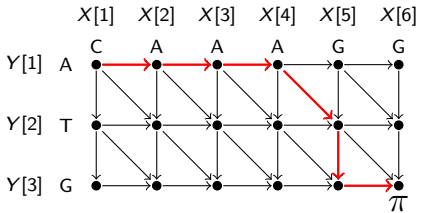
Pattern Matching Under Dynamic Time Warping Distance

SPIRE'22



Anne Driemel, Pierre Peterlongo, Tatiana Starikovskaya

Formal definition of $\mathrm{DTW}(X, Y)$ and dynamic programming

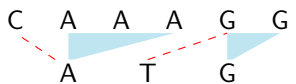


$$\mathrm{cost}(\pi) = \sum_{(i,\ j)\in\pi} d(X[i], Y[j])$$

$$\mathrm{DTW}(X, Y) = \min_{\pi} \mathrm{cost}(\pi)$$

s.t. π goes from top left to bottom right.

Path correspondance to alignment



$$\mathrm{DTW}(\mathrm{CAAAG},\mathrm{ATG}) = 2$$

Comparison to the edit distance

$$D[i,j] = \min\{\underbrace{D[i-1,j-1]}_{\text{top-left}}, \underbrace{D[i-1,j]}_{\text{top}}, \underbrace{D[i,j-1]}_{\text{left}}\} + d(X[i], Y[j])$$

$$ED[i,j] = \min\{\underbrace{ED[i-1,j-1]}_{\text{top-left}} + d(X[i], Y[j]), \underbrace{ED[i-1,j]}_{\text{top}} + 1, \underbrace{ED[i,j-1]}_{\text{left}} + 1\}$$

DTW vs ED

$\mathrm{ED}(AAAATG, AATC) = 3$ whereas $\mathrm{DTW}(AAAATG, AATC) = 1$.
$\mathrm{DTW}(AAAATG, AATCCC) = 3$

Our contributions:

For T and P two strings, $|T| = N$ and $|P| = M$, $|\mathrm{RLE}(T)| = n$ and $|\mathrm{RLE}(P)| = m$.
Pattern matching under DTW distance:

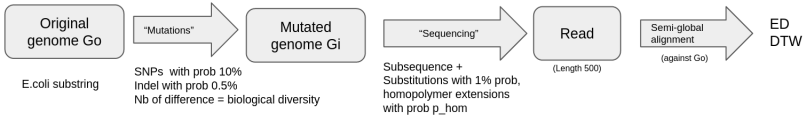
- $\mathcal{O}(Nm + nM)$-time general algorithm which can for an integer distance, compute all values bellow k in $\mathcal{O}(nmk)$-time.
- Toy implementation available on github.
- An $\mathcal{O}(L^{\varepsilon})$-approximation, for any $0 < \varepsilon < 1$, in $\mathcal{O}(L^{1-\varepsilon} \cdot mn \log^3 L)$-time with $L = \max(N, M)$.
- (A $\mathcal{O}(n + m)$-time algorithm for $k = 1$.)

Open questions:

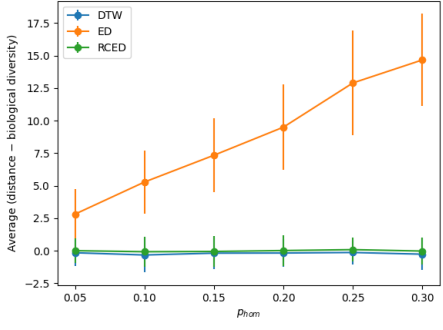
- Is a $\mathcal{O}(k(n + m))$-time algorithm possible?
- Can those improvements benefit applications?

Experiments: visualization on simulated data

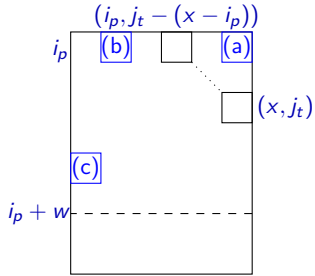
Problem: How to design a protocol that isn't biased towards ED?
Model to illustrate the impact of homopolymers on ED.



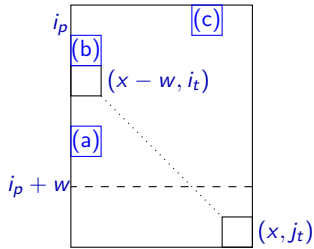
We compare the values of the two distances as the probability of extending homopolymers increases.



Border Formula



Case 1: $x - i_p \leq w$



Case 2: $x - i_p > w$

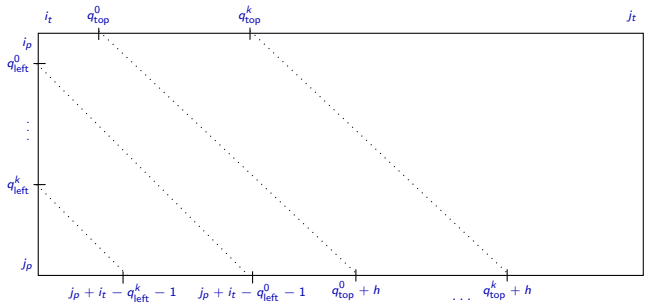
For a block $D[i_p .. j_p, i_t .. j_t]$ let $h = j_p - i_p$, $w = j_t - i_t$, and $d = d(P[i_p], T[i_t])$.
We have for every $i_p < x \leq j_p$:

$$D[x, j_t] = \begin{cases} D[i_p, j_t - (x - i_p)] + (x - i_p) \cdot d \text{ if } x - i_p \leq w; \\ D[x - w, i_t] + w \cdot d \text{ otherwise.} \end{cases}$$

This formula implies a $\mathcal{O}(Nm + Mn)$-time algorithm.

Low-distance regime for integer distances

We can just represent the values $\leq k$ in a compact format: For ℓ an integer, q^{ℓ}_{top} is the largest position such that $D[i_p, q^{\ell}_{\text{top}}] \leq \ell$.



Computing q^{ℓ}_{bot} and q^{ℓ}_{right} from q^{ℓ}_{top} and q^{ℓ}_{left} can be done in $\mathcal{O}(k)$ time.
The compact representation of the last row can be computed in $\mathcal{O}(nmk)$ time.

Optimal Square Detection Over General Alphabets

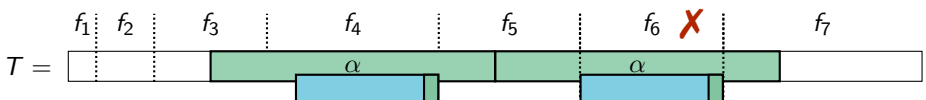
SODA'23


Jonas Ellert, Paweł Gawrychowski

The use of Lempel-Ziv factorization to detect squares

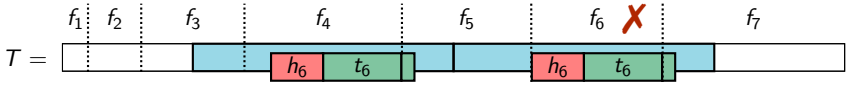
We process T left to right, square testing, we can note that:

- We don't have to look for a square fully included in $f_i[.. - 1]$.
- If the previous occurrence of $f_i[.. - 1]$ overlaps with itself, then it implies a square.
- The right arm of square can overlap at most two phrases.
- For x and y square-free string we can detect squares in xy with their right arm in y contained in $\mathcal{O}(|y|)$ time.

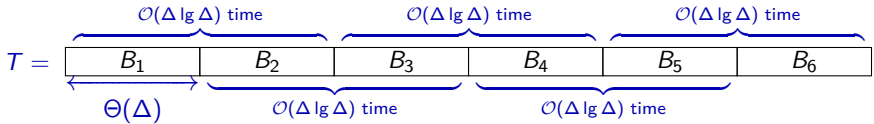


Building the factorization requires $\Omega(n\sigma)$ GUA comparisons... :(

Using the Δ-approximate factorization to detect squares



- The right arm of square intersects at most two factors.
- Squares that are larger than $\geq 8\Delta$ intersect at least a tail of length $\geq \Delta$.
- We can use the Main+Lorentz'84 to detect in $\mathcal{O}(n)$ time any such squares.
- For squares $\leq 8\Delta$, we slice the text into overlapping small blocks.



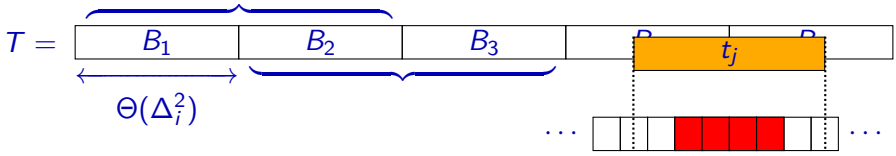
$\Longrightarrow$ Choose $\Delta = (\sigma \lg n)^2$ to achieve $\mathcal{O}(n(\lg \sigma + \lg \lg n))$ time. But we cannot know σ...

We proceed in $\log \log n$ phases, with decreasing Δ_i, detecting "long" squares until we see that the alphabet is too large and we can "afford" $\mathcal{O}(n \log \Delta_i)$ + Amortization across the levels.

Estimating the alphabet size

- proceed in $\mathcal{O}(\lg\lg n)$ phases, with $\Delta_1 = \Theta(\sqrt{n})$ and $\Delta_{i+1} = \sqrt{\Delta_i}$
- in phase i, try to detect square of length $\Omega(\Delta_i)$ and $\mathcal{O}(\Delta_{i+1}) = \mathcal{O}(\Delta_i^2)$

$\mathcal{O}(\Delta_i^2 + \frac{(\Delta_i)^2 \cdot \lg \Delta_i \cdot \sigma}{\sqrt{\Delta_i}})$ time, which is $\mathcal{O}(\Delta_i^2)$ if $\sigma = \mathcal{O}(\sqrt[4]{\Delta_i})$



$\implies \mathcal{O}(n \lg \sigma)$ time

- single phase takes $\mathcal{O}(n)$ time
- if $\sigma = \Omega(\sqrt[4]{\Delta_i})$, then $\mathcal{O}(\Delta_i^2 \lg(\Delta_i^2)) = \mathcal{O}(\Delta_i^2 \lg \sigma)$

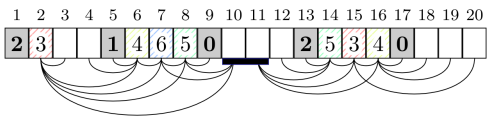
  $\implies$ run **Main+Lorentz'84** on each block pair, finish in $\mathcal{O}(n \lg \sigma)$ time

  $\implies$ total time $\mathcal{O}(n(\lg \sigma + \lg\lg n))$ without knowing σ

Lowerbound on square detection

Player vs Adversary: aims at giving as little information as possible

- Keep track of the comparisons asked through a conflict graph.
- Divide the string in blocks of $\sigma/4$.
- Separate the blocks by character of the Prouet-Thue-Morse square-free sequence.
  $\Longrightarrow$ square-free iff all blocks are square free.
- **Coloring rule**: If a node/position reaches degree $\sigma/4$, we color it by avoiding the color of the neighbours + the colors of the same block.



Analysis of the conflict graph after the player terminates.

The Longest Common Substring with Approximately k Mismatches

CPM'20



Tomasz Kociumaka, Jakub Radoszewski, Tatiana Starikovskaya

The Longest Common Substring with Approximately k mismatches

$LCS_{\tilde{k}}$

Input: Two strings X, Y of length n, an integer k.
Output: A substring of X of length at least LCS_k that occurs in Y with at most $(1+\varepsilon)\cdot k$ mismatches.

[Kociumaka, Radosweski, Starikovskaya'19]

$LCS_{\tilde{k}}$ can be solved for $\varepsilon < 2$, in $\mathcal{O}(n^{1+1/(1+\varepsilon)}\log^2 n)$ time and $\mathcal{O}(n^{1+1/(1+\varepsilon)})$ space.

Our contributions

Upper bounds

Let $\varepsilon > 0$ be an arbitrary constant. The LCS with Approximately k Mismatchesproblem can be solved correctly with high probability:

1) in $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)} \log^2 n)$ time and $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$ space assuming a constant-size alphabet;
2) and $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^3 n)$ time and $\mathcal{O}(n)$ space for alphabets of arbitrary size.

Lower Bound

Assuming SETH, for every constant $\delta > 0$, there exists a constant $\varepsilon = \varepsilon(\delta)$ such that given X and Y of size n computing the LCS with Approximately k Mismatchesrequires $\Omega(n^{2-\delta})$ time.

The Longest Common Substring problem

LCS

Input: Two strings X, Y of length n.
Output: The longest substring that occurs both in X and Y.

Issue: It is not robust.

$$X = a^{2m+1} \text{ and } Y = a^{2m}b \Rightarrow LCS(X, Y) = 2m$$
$$X = a^m b a^m \text{ and } Y = a^{2m}b \Rightarrow LCS(X, Y) = m$$

Only one character changed, and the LCS has been divided by 2.

The Longest Common Substring with k mismatches

LCS with k Mismatches

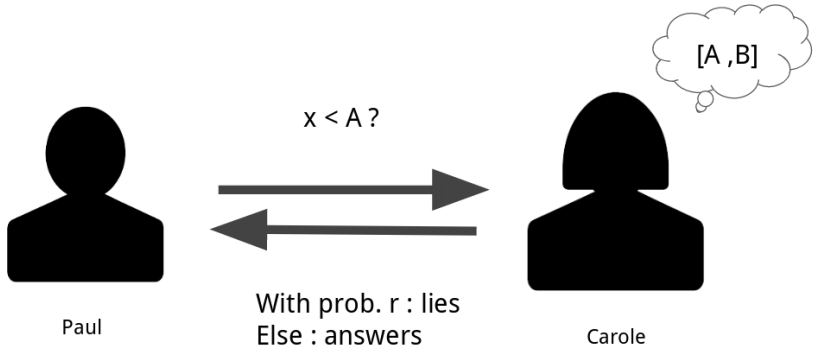
Input: Two strings X, Y of length n, an integer k.
Output: A substring of X that occurs in Y with at most k mismatches.

However...

[Kociumaka, Radoszweski, Starikovskaya'19]

There is a $k = \Theta(\log(n))$ such that LCS_k can't be computed in strongly subquadratic time, unless SETH is false.

Twenty question game with a liar



[Dhagat, Gács, Winkler '92]

For $A = B$, Paul has a winning strategy for all $r < \frac{1}{3}$ asking $Q = \lceil \frac{8 \log N}{(1-3r)^2} \rceil = \Theta(\log n)$ questions.

Decision variant

Input: integers k, ℓ, a constant $\varepsilon > 0$, strings S_1, S_2 of length n
Output:

1. YES if $\ell \leq \mathrm{LCS_k}$;
2. YES or NO if $\mathrm{LCS_k} < \ell \leq \mathrm{LCS}_{(1+\varepsilon)\mathrm{k}}$;
3. NO if $\mathrm{LCS}_{(1+\varepsilon)\mathrm{k}} < \ell$.

The answer must be correct with probability at least $3/4$.

Longest Common Substring with approx. k mismatches:

- $A = \mathrm{LCS_k}$ and $B = \mathrm{LCS}_{(1+\varepsilon)\mathrm{k}}$.
- An algorithm for the decision variant plays the role of Carole.
- With $\lceil \frac{8 \log n}{(1-3r)^2} \rceil$ questions, Paul will find $x \in [\mathrm{LCS_k}, \mathrm{LCS}_{(1+\varepsilon)\mathrm{k}}]$ for some $1/4 < r < 1/3$.

Locality-Sensitive Hashing

Nearest Neighbour search, data clustering...
In string processing: Andoni and Indyk for a space-efficient randomized index for approximate pattern matching.

1. Projections: $h(S) = S[a_{p_1}]S[a_{p_2}]\cdots S[a_{p_m}]$

   And Collisions-Sets (Karp-Rabin fingerprints).

$$C_\ell^{\mathcal{H}} = \{(S_1, S_2, h) : \varphi(h(S_1 0^{n-\ell})) = \varphi(h(S_2 0^{n-\ell}))\}$$

2. Dimension reduction. With probability at least $1 - n^{-\beta}$, for all $u, v \in P$:
   1) if $\|\mathrm{sk}_\alpha(u) - \mathrm{sk}_\alpha(v)\|^2 \leq (1+\alpha)\cdot k$, then $d_H(u,v) \leq (1+\alpha)\cdot k$;
   2) if $\|\mathrm{sk}_\alpha(u) - \mathrm{sk}_\alpha(v)\|^2 > (1+\alpha)\cdot k$, then $d_H(u,v) \geq k$.

Algorithm

1: Choose a set $\mathcal{H}$ of $\Theta(n^{1/(1+\varepsilon)})$ functions from Π^m u.a.r.
2: $C_l^{\mathcal{H}} :=$ set of all collisions of l-length substrings of S_1, S_2 under the hash functions in $\mathcal{H}$
3: Draw a collision $(X, Y) \in C_\ell^{\mathcal{H}}$ uniformly at random
4: **if** $\mathrm{Ham}(X, Y) \leq (1+\varepsilon) \cdot k$ **then return** YES
5: Choose a subset $C' \subseteq C_l^{\mathcal{H}}$ of size $\min\{C_\ell^{\mathcal{H}}, 4nL\}$
6: **for** $(X, Y) \in C'$ **do**
7: **if** $\mathrm{Ham}(S_1, S_2) \leq k$ **then return** YES
8: **return** NO

Running time $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n)$:

1. Compute the hash values and C': $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n)$ time (FFT)
2. Pick a random collision: $\mathcal{O}(n^{1+1/(1+\varepsilon)})$ time (reservoir sampling)
3. Test in line 5: $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$ time (dimension reduction)
4. Test in line 7: $\mathcal{O}(n)$ time (character-by-character)

Experiments

None of the previous solutions have been implemented.
The only algorithm that seemed to be practical enough is the dynamic programming one [Flouri et al.'15]

We compared our algorithm with the dynamic programming one

- On random strings;
- On strings extracted from E. coli.

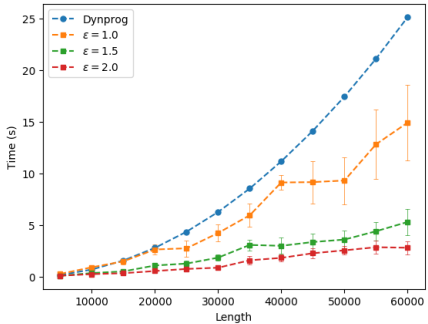
Lengths from 5000 to 60000, $k = 10, 25, 50$

Adjustments to the theory

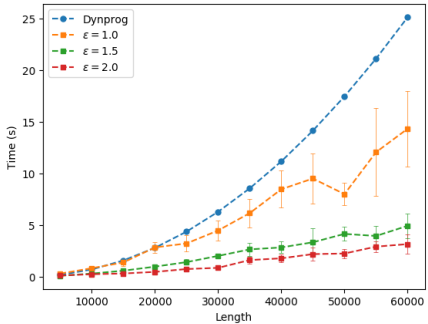
Implemented in C++ available on github.

1. Sketching for the Hamming distance via dimension reduction: replaced by a naive comparison character by character, Bit parallelism.
2. Computation of the collisions: A naive implementation, FFT and NTT.
3. The twenty question game: the twenty question game, binary search.
4. The number of hash functions L: $L = n^{1/(1+\varepsilon)}$, $L = n^{1/(1+\varepsilon)}/\log(n)$.

Running time



(a) Random, $k = 25$



(b) E. coli, $k = 25$

- For each length, we performed 10 independent experiments
- Big standard deviation for $\varepsilon = 1$, negligible for $\varepsilon = 1.5$ and $\varepsilon = 2.0$
- Gain up to a factor of 10 on strings of length 60000

Distortion and accuracy

We estimate distortion by computing two values:
$r_{\min}(\varepsilon, k) = \min_{S_1,S_2}(\mathrm{LCS}_{\tilde{\mathrm{k}}}(S_1, S_2)/\mathrm{LCS}_{\mathrm{k}}(S_1, S_2))$
$r_{\max}(\varepsilon, k) = \max_{S_1,S_2}(\mathrm{LCS}_{\tilde{\mathrm{k}}}(S_1, S_2)/\mathrm{LCS}_{\mathrm{k}}(S_1, S_2))$
Furthermore, we can only err by returning a string shorter than $\mathrm{LCS}_{\mathrm{k}}$.

	Random					
	$\varepsilon = 1.0$		$\varepsilon = 1.5$		$\varepsilon = 2.0$	
$k = 10$	0.92	1.50	1.00	1.53	1.13	1.87
	err = 7%		err = 0%		err = 0%	
$k = 25$	1.10	1.48	1.30	1.70	1.55	2.11
	err = 0%		err = 0%		err = 0%	

	E. coli					
	$\varepsilon = 1.0$		$\varepsilon = 1.5$		$\varepsilon = 2.0$	
$k = 10$	0.86	1.41	0.91	1.47	0.95	1.71
	err = 34%		err = 13%		err = 8%	
$k = 25$	0.94	1.45	0.96	1.75	0.98	1.96
	err = 7%		err = 5%		err = 2%	

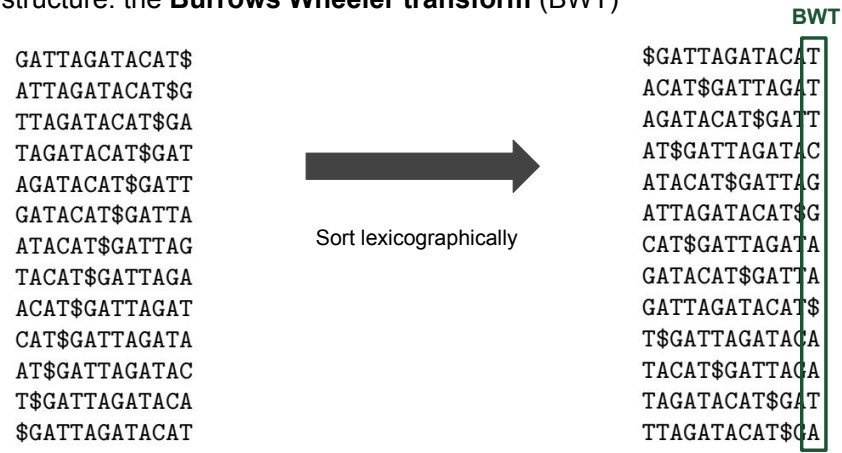
XBWT Indexing of Aligned Readsets

WABI'21

Travis Gagie, Giovanni Manzini

Compression: The Burrows Wheeler Transform

A classical structure: the **Burrows Wheeler transform** (BWT)



Self Indexing: The FM-index

Searching for GAT in GATTAGATACAT:

- **Count**(P) in **O(|P|)**
- **Locate**(P) in **O(|P| + occ)**

Applied in short read aligners such as **Bowtie** and **BWA** !

	F	L
0	$GATTAGATACAT	
1	ACAT$GATTAGAT	
2	AGATACAT$GATT	
3	AT$GATTAGATAC	
4	ATACAT$GATTAG	
5	ATTAGATACAT$G	
6	CAT$GATTAGATA	
7	GATACAT$GATTA	
8	GATTAGATACAT$	
9	T$GATTAGATACA	
10	TACAT$GATTAGA	
11	TAGATACAT$GAT	
12	TTAGATACAT$GA	

Count(GAT)=2

3rd to 5th occ of A

Self Indexing: Runs

The BWT can be quite repetitive:

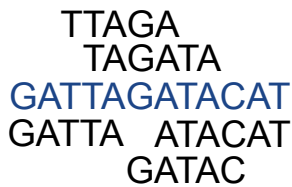
BWT(GATTAGATACAT$) = TTTCGGAA$AATA

A **run** is a **substring of identical letters**, for example AAABBBA has 3 runs.

BWT(GATTAGATACAT$) = TTTCGGAA$AATA => 8 runs

Method	Space	Count(P)	Locate(P)
The Run-Length FM index **[Mäkinen, Navarro, Siren, and Välimäki.]**	**O(r)**	**O(|P|log|T|)**	**/ (not in O(r))**
The r-index **[Gagie, Navarro, and Prezza.]**	**O(r)**	**O(|P|)**	**O(|P|+occ)**

Indexing of a collection: read sets



- Could we have a similar structure with space **depending on the number of runs** for an aligned readset?
- Can we find a link between the **number of runs of that similar structure for the readsets** and the **runs in the BWT of the genome** it is aligned to?

Indexing of a collection: Naive approach

Concatenate the reads with a separator “$” and build the FM-index of the entire string.

Example:

S = GATTA$TTAGA$TAGATA$GATAC$ATACAT$

BWT(S) = CATAATGTTTTTCGG$GAAAA$$AATAATA => 20 runs.

Issues:

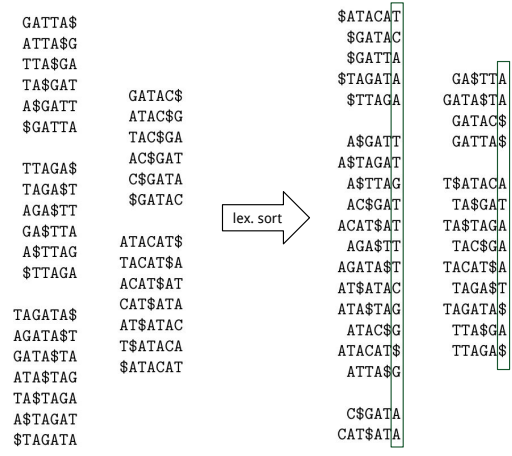
- Computing the BWT for such a long string is challenging (and the context before the dollars is not relevant).
- The $ break some runs as in CGG$G in the example.

Indexing of a collection: The EBWT

Extended BWT: lexicographically sort all rotations of each string.
[Mantaci, Restivo, Rosone, and Sciortino, An extension of the Burrows–Wheeler Transform, 2007]

Example:
EBWT(S) =
TCAAATTGTTTTCGG$GAAAA$$ATAAATA
=> 19 runs.

- Easier to build and update than the naive approach.
- Still has more than double the number of run than BWT(GATTAGATACAT).



Indexing of a collection: the heuristics on order

RLO: Reorganize the reads in the reversed lexicographic order (co-lexicographic order)

Example:

RLO(S) = {GATTA$, TTAGA$, TAGATA$, GATAC$, ATACAT$}
EBWT(RLO(S)) = AAACTGTTTTTTCGG$GAAAA$$AATAATA => 19 runs

SPRING:

[SPRING: a next-generation compressor for FASTQ data, Chandak, Tatwawadi, Ochoa, Hernaez, and Weissman, 2018]

Attempts to reorder reads according to their position in the genome.

EBWT(SPRING(S))= ACATATTGTTTTCGG$GAAAA$$ATAAATA => 22 runs

Indexing of a tree: The XBWT

eXtended BWT: Generalization of the BWT for labeled trees where nodes are sorted by their label from the node to the root and we output the outgoing edges.

XBWT(T) = ab abc $c $ $ $ a ac a a $ $

(Can be seen as a sub-case of a Wheeler graph.)

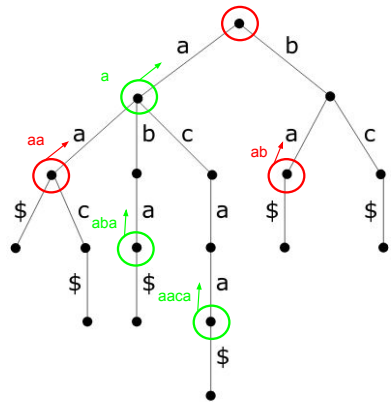


Figure: XBWT Tricks, *Giovanni Manzini*

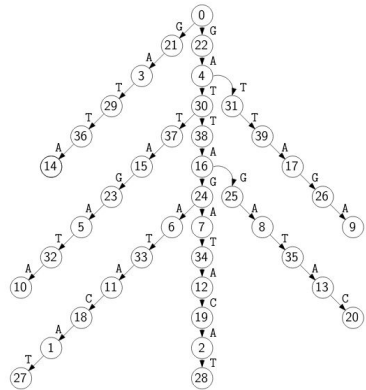
Our approach: using alignment and XBWT

If the reads correspond to a known assembled genome, use the genome as additional context for better compression.

Example:

TTAGA
TAGATA
GATTAGATACAT
GATTA ATACAT
GATAC

T =

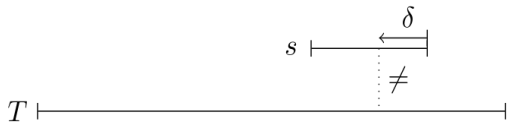


XBWT(T)=GGTTTTTTTTTCCCGGGGAAAAAAAAATTTTAAAAAAAA => 7 runs.

Our approach: Theoretical guaranties

If we create such a tree, where the **reads are errorlessly sampled and aligned to the reference**, then the XBWT of the tree has the **same number of runs** as the BWT of the reverse of the reference.

But **in reality** the reads are **not perfectly matching** the assembled genome ..?



If the reads differ from the reference string and that the average **distance from first difference** (insertion, deletion, or substitution) **to the end** of the read is **δ**, then, the XBWT of the tree will have at most **2δ additional runs** per reads.

Our approach: Theoretical guaranties

If we create a labeled tree T as explained, let:
r be the number of runs in the XBWT,
t be the number of reads,
Then in **O(r+t)** words of space,
We can:
Count(P) in **O(|P|log log(|T|))**
Locate(P) in **O((|P|+occ)log log(|T|))**

What are the runs in the XBWT in practice ? And how do we build the XBWT ?

For scalability: prefix-free parsing construction

The reference and genomes are typically tens of Gb of data…
Consequently, **algorithms designed to work in RAM may not be practical**.
We chose to adapt a previous technique **Prefix free parsing.**

Prefix free construction takes advantage of the **highly repetitive** nature of genomic databases using **context-triggered piecewise hashing**.
[Boucher, Gagie, Kuhnle, Langmead, Manzini and Mun, Prefix-free parsing for building big BWTs, WABI 2017]

[PFP Data Structure, Boucher, Cvacho, Gagie, Holub, Manzini, Navarro, Rossi, 2020]
54 GB peak memory for 1000 variants of human chromosome 19, initially occupying roughly **56 GB**.

To aim for a scalable structure **we adapted Prefix free parsing to build the XBWT**.

Experiments: comparison to the state of the art

A preliminary comparison only on the number of runs for now.

We compared to:

- **EBWT** (using the ropebwt2 implemetation), with and without $.
- **SPRING + EBWT,** with and without $.
- **RLO + EBWT,** with and without $.

Removing the $ reduced the number of runs between **2.7%** and **29.2%**.

Experiments: datasets and protocol

Only reads matched to the genome, not to the reverse complement.

Reads and corresponding genome:

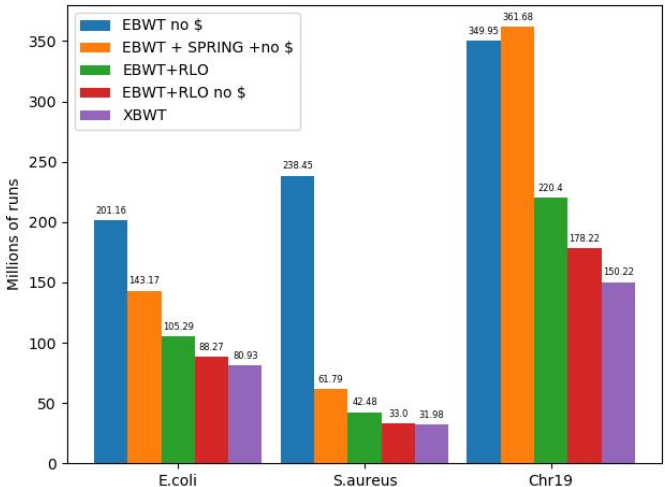
- **E.coli**: from the single cell dataset
- **S.aureus**: from the single cell dataset
- **R.sphaeroides**: from the Gage-b dataset

Reads aligned to a reference genome:

- **Chr19**: a reference genome and the reads of a HiSeq 2000 readsets that aligned

bwa-mem used to align the reads to the genomes.

Experiments: results



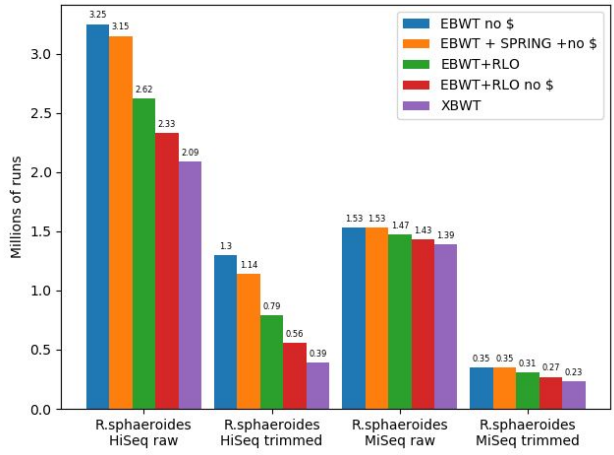
Dataset	Nb. reads	Coverage	δ	errorless reads
E.coli	14M	304×	13	57%
S.aureus	26M	927×	7	88%
Chr19	34M	57×	15	71%

- RLO+EBWT has much less runs than the plain EBWT.
- **XBWT performs best (but RLO is close)**
- On S.aureus, RLO, RLO no $ and XBWT are very close.

Experiments: Gage-b R.sphaeroides

Dataset	Number of reads	Read length	Coverage
HiSeq			
raw	166 820	101	46×
trimmed	134 207	up to 101	37×
MiSeq			
raw	23 102	251	24×
trimmed	20 046	up to 251	20×

Dataset	δ	Errorless reads	Error rate
HiSeq			
raw	27	31.34%	0.04%
trimmed	6	83.26%	0.01%
MiSeq			
raw	122	0.25%	0.15%
trimmed	29	63.55%	0.03%



Summary of our contributions:

- Looking at the genome for additional context for better compression is worth investigating !
  - We provide theoretical time and space guaranties depending on the number of reads and the number of runs.
  - We show an upper bound on the number of runs depending on the errors in the reads compared to the genome.
  - The experimental number of runs is comparatively small.
- Prefix-free construction of the BWT can be adapted for the XBWT.
- A similar approach could be used to improve the space usage of the hybrid index. (Not explored in this talk)

//TODO:

- Larger scale analysis (on human genome, on long reads)
- FM-index, implementation and time analysis
- Time comparison of PFP construction of the XBWT compared to other construction

  [BWT-tunneling by Uwe Baier, Wheeler sort by Jarno Alanko]

Gapped Consecutive Matching

1. Grammar Compressed Indexing (CPM'23)
2. Grammar Compressed Pattern Matching



Paweł Gawrychowski, Tatiana Starikovskaya, Teresa Anna Steiner

Indexing for complex queries

Indexing: preprocess a text or a collection of texts into a data structure that allows locating occurrences of a query pattern in the texts.

- If the query is a string, multiple space- and time-efficient solutions exist

However, *it is desirable to allow for* ***more general queries!***

- For regular expression patterns, there cannot be a data structure with polynomial-time preprocessing and sublinear query time, conditioned on the online matrix-vector multiplication conjecture [Thankanchan and Gibney, 2021]
- What about simpler models with just two patterns?

Indexing for complex queries

Indexing: preprocess a text or a collection of texts into a data structure that allows locating occurrences of a query pattern in the texts.

- If we are looking for all texts in a collection that contains two string patterns P_1, P_2, or all texts containing P_1 but not P_2, the asymptotically fastest linear-space solutions use $O(\sqrt{N})$ query time, where N is the total length of the texts [Hon et al. 2010, Hon et al. 2012]
- This was shown to be optimal conditioned on Boolean Matrix Multiplication [Larsen et al. 2014] and on the 3SUM conjecture [Kopelowitz et al. 2016]

Indexing for complex queries

Indexing: preprocess a text or a collection of texts into a data structure that allows locating occurrences of a query pattern in the texts.

- [Kopelowitz and Krauthgamer 2016] considered the problem of retrieving the pair of closest occurrences of two patterns P_1, P_2 in a text T.
- For a text of length N, they showed an index using space $\tilde{O}(N^{1.5})$ with $\tilde{O}(N\sqrt{N})$ preprocessing time and $\tilde{O}(|P_1| + |P_2| + \sqrt{N})$ query time.
- By establishing a connection with Boolean Matrix Multiplication, they highlighted a difficulty in removing the $\sqrt{N}$ factor both from the preprocessing and the query time.

$\tilde{O}$ hides the logarithmic factors.

Gapped consecutive indexing

Gapped indexing for consecutive occurrences: preprocess an text T of length N into a data structure, which allows, given a range $[a, b]$ and two patterns P_1, P_2, to retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

Gapped consecutive indexing

Gapped indexing for consecutive occurrences: preprocess an N-length text T into a data structure, which allows, given a range $[a, b]$ and two string patterns P_1, P_2, to retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

- [Navarro and Thankanchan 2016] For the case $P_1 = P_2$, $O(N \log N)$-space index with $O(|P_1| + |P_2| + \text{occ})$ query time.
- [Bille et al. 2021] For the general case $P_1 \neq P_2$, no $\tilde{O}(N)$-space index can achieve $\tilde{O}(|P_1| + |P_2| + \sqrt{N})$ query time conditioned on the Set Disjointness conjecture.

For highly compressible texts, can we design an efficient index for this problem?

$\tilde{O}$ hides the logarithmic factors.

Choice of compression method

The answer, of course, depends on the chosen compression method…

- We assume that the text is represented by a straight-line program (SLP), which is a context-free grammar describing exactly one string.

Example: The SLP $\{A \to BC, B \to ba, C \to DD, D \to na\}$ generates $banana$.

- SLPs are **capable of describing strings of exponential length** (in the size of the representation).
- Capture the popular **Lempel-Ziv compression method** up to a log factor.
- On the other hand, SLPs provide a convenient interface, allowing e.g. for efficient random access [Bille et al. 2015].

Indexing in compressed space

Assuming that a string T of length N is described by an SLP with g productions, there are multiple $\tilde{O}(g)$-space indexes for **classic pattern matching**:

Space	Query time	Reference
$O(g)$	$O(m \log\log N + \mathrm{occ} \log g)$	Claude and Navarro 2012
$O(g \log N)$	$O((m + \mathrm{occ})\log g)$	Claude et al. 2021
$O(g \log N)$	$O(m + \mathrm{occ} \log^{\epsilon} N)$	Christiansen et al. 2021
$O(g \log N)$	$O((m \log m + \mathrm{occ})\log g)$	Díaz-Domínguez et al. 2021

Lower bounds for compressed data

Some problems cannot avoid a high dependency on the size of an uncompressed string:

- Pattern matching with wildcards [Aboud et al. 2017]
- Longest common subsequence [Aboud et al. 2017]
- Median edit distance [Kociumaka et al. 2022]
- Center edit distance [Kociumaka et al. 2022]

What about consecutive pattern matching?

Our results

Grammar compressed indexing for gapped consecutive occurrences: preprocess an N-length text T given as a grammar of size g into a data structure, which allows, given a range $[a, b]$ and two string patterns P_1, P_2, of length m to retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

Cases	Space	Query time
unbounded (a = 0, b = N)	$O(g^2 \log^4 N)$	$O(m \log N + (1 + \mathrm{occ}) \cdot \log^3 N \log\log N)$
a = 0	$O(g^5 \log^5 N)$	$O(m \log N + (1 + \mathrm{occ}) \cdot \log^4 N \log\log N)$

We obtain those results by using **locally consistent grammars** !

Our results

Gapped indexing for consecutive occurrences:
preprocess an N-length text T into a data structure, which allows, given a range $[a, b]$ and two string patterns P_1, P_2, of length m to retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

Cases	Space	time
unbounded (a = 0, b = N)	$O(g^2 \log^4 N)$	$(1 + \text{occ}) \cdot \log^3 N \log\log N)$
a = 0	$O(g^5 \log^5 N)$	$O(m \log N + (1 + \text{occ}) \cdot \log^4 N \log\log N)$

Achieving $\tilde{O}(g)$ space and $\tilde{O}(m + \text{occ})$ query time would contradict the lower bound of Bille et al. 2021

We obtain those results by using **locally consistent grammars** !

Our results

Gapped indexing for consecutive occurrences:
preprocess an N-length text T into a data structure, which allows, given a range $[a, b]$ and two string patterns P_1, P_2, of length m [illegible] all pairs of consecutive occurrences of P_1, P_2 separat[illegible] $b]$.

In the unbounded case, it might be possible to achieve $\tilde{O}(g)$ space and $\tilde{O}(m + \mathrm{occ})$ query time

Cases	Space	[illegible]uery time
unbounded (a = 0, b = N)	$O(g^2 \log^4 N)$	$O(m \log N + (1 + \mathrm{occ}) \cdot \log^3 N \log\log N)$
a = 0	$O(g^5 \log^5 N)$	$O(m \log N + (1 + \mathrm{occ}) \cdot \log^4 N \log\log N)$

We obtain those results by using **locally consistent grammars** !

Run-length SLP

Run-length SLP a set of non-terminals, a set of terminals, an initial symbol, and a set of productions, such that:

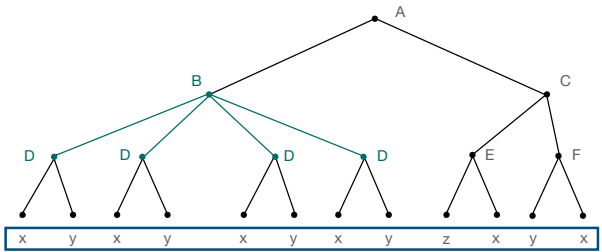
- Each production has form $A \rightarrow BC$ or $A \rightarrow B^k$, where A is a non terminal and B, C can be either terminals or non-terminals.
- Every non-terminal is on the left-hand side of exactly one production (=> it generates exactly one string).

Expansion(S), is the string “generated” by the non-terminal S.

The string obtained by iterative replacement of non-terminals by the right-hand sides of the production rules, until only terminals remain.

A run-length SLP describes the expansion of its initial symbol.

Run-length SLP

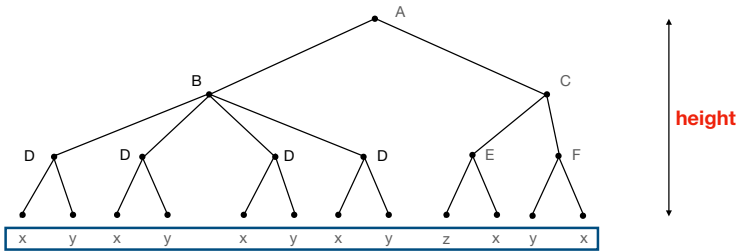


The parse tree of a run-length SLP

non-terminals $= \{A, B, C, D, E, F\}$ and terminals $= \{x, y, z\}$

productions: $A \to BC$, $B \to D^4$, $D \to xy$, $C \to EF$, $E \to zx$, $F \to yx$

Run-length SLP



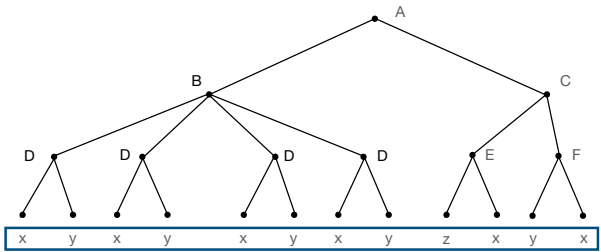
The parse tree of a run-length SLP

non-terminals $= \{A, B, C, D, E, F\}$ and terminals $= \{x, y, z\}$

productions: $A \to BC$, $B \to D^4$, $D \to xy$, $C \to EF$, $E \to zx$, $F \to yx$

size

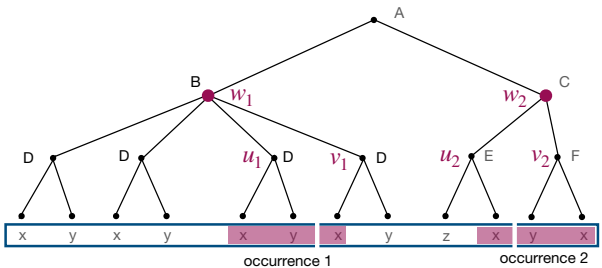
SLP to a locally-consistent run-length SLP



Corollary of [Gawrychowski et al. 2018]

There is a Las-Vegas algorithm that converts an SLP of size g describing a string T of length N into a **locally-consistent** run-length SLP of size $O(g \log N)$ and height $O(\log N)$ describing the same string T in $O(g \log N)$ time.

SLP to a locally-consistent run-length SLP



Locally-consistent run-length SLP

There are only $O(\log N)$ possible splits of P, and we can compute them in $O(|P|\log N)$ time

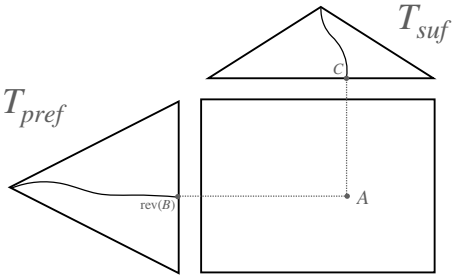
Definition: Split

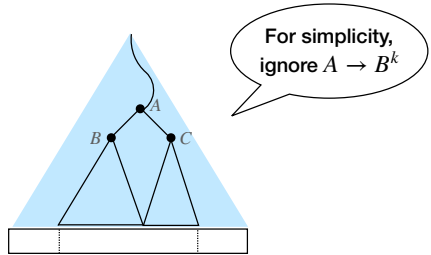
For an occurrence $T[\ell, r]$ of a pattern P, let w be the lowest node of a parse tree containing it, we say that $T[\ell, r]$ is **relevant** for the label of w (a non-terminal).

$T[\ell, r]$ is **split at position** i if there exist children u, v of w such that $T[\ell, \ell + i]$ is contained in u and $T[\ell + i + 1, r]$ in v.

Example: Occurrence 1 xyx is split at position 2 and relevant for B, occurrence 2 is split at position 1 and relevant for C.

Why is local consistency interesting?

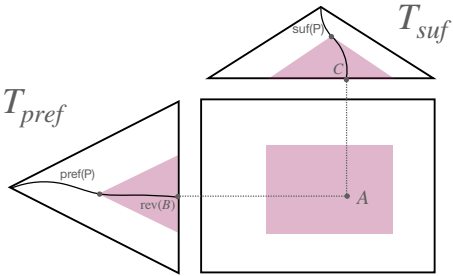


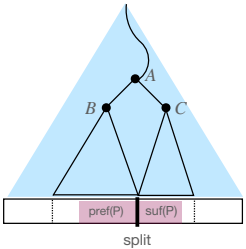


…because we can search for relevant occurrences quickly using the following structure:

- For every $A \to BC$, add $\mathrm{rev}(\mathrm{expansion}(B))$ to T_{pref} and $\mathrm{expansion}(C)$ to T_{suf}
- Create a point (r_B, r_C) (the lexicographic rank of the expensions) for every $A \to BC$
- Build an orthogonal range data structure on the points

Why is local consistency interesting?





To find relevant occurrences of a pattern P in non-terminals:

- For each split s, search for $\mathrm{pref}(P) = \mathrm{rev}(P[1..s])$ in T_{pref} to obtain an interval I_{pref} of leaves starting with it, and for $\mathrm{suf}(P) = P[s+1..]$ in T_{suf} to obtain an interval I_{suf}
- Report all non-terminals in $I_{pref} \times I_{suf}$

We show an even stronger result

For a run-length SLP representing a string T of length N, with size g and height $O(\log n)$,

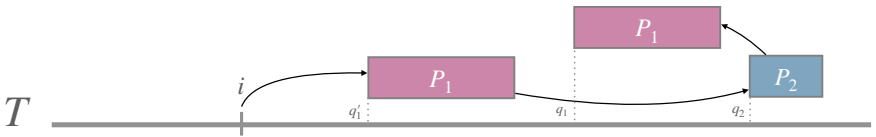
There is **a $O(g^2 \log^2 N)$-space data structure** that preprocesses an m-length pattern P in $O(m \log N + \log^2 N)$ time and can, answer the following queries in $\mathrm{polylog}\ N$ time:

For a given non terminal A, in $\mathrm{expansion}(A)$,

- Report relevant occurrences of P;
- Decide whether there is an occurrence of P;
- Report the leftmost/rightmost occurrence of P ;
- Find a predecessor/successor occurrence of P given a position q.

Corollary: unbounded case

Task: report all consecutive occurrences of P_1, P_2 in a N-length string T described by a run-length SLP of size g and height $\log N$.



1. Find the leftmost occurrence q_1' of P_1 in $T[i\,..\,]$ (successor)
2. Find the leftmost occurrence $q_2 \geq q_1'$ of P_2 (successor)
3. Find the rightmost occurrence $q_1 \leq q_2$ of P_1 (predecessor)
4. Report (q_1, q_2) and set $i = q_2 + 1$

Time $\tilde{O}(m + (\mathrm{occ} + 1)\mathrm{polylog}\, N)$**, space** $\tilde{O}(g^2)$**.**

Idea of our index for the case $a = 0$

Task: given an N-length string T described by a run-length SLP of size g and height $\log N$, report all consecutive occurrences of P_1, P_2 in T separated by distance in $[0,b]$.

- For each non-terminal of the grammar, retrieve relevant consecutive occurrences separated by distance in $[0,b]$.
- Generate all consecutive occurrences separated by distance in $[0,b]$ by traversing a pruned parse tree of the grammar (using a standard technique borrowed from classic pattern matching compressed-space indexes).

Summary

Complex matching: **Gapped consecutive occurrences:** given a range $[a, b]$ and two string patterns P_1, P_2, retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

Sketch as input: Grammar compressed input (local consistency preserves splits)

Indexing: preprocess a text T of length N given as grammar g into a data structure.

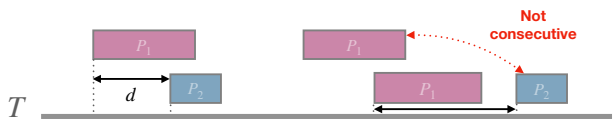
Case	Space	Query time
unbounded (a = 0, b = N)	$O(g^2 \log^4 N)$	$O(m \log N + (1 + \text{occ}) \cdot \log^3 N \log\log N)$
a = 0	$O(g^5 \log^5 N)$	$O(m \log N + (1 + \text{occ}) \cdot \log^4 N \log\log N)$

Can better space be achieved? Solution for the general case?

Is the dual problem of consecutive compressed pattern matching easier?

Grammar compressed consecutive pattern matching

Dual problem: Given T of length N as grammar of size g, a range $[a, b]$, and two string patterns P_1, P_2, retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$. **Process the text and the patterns at the same time !**



If T is given uncompressed: we can just go from left to right, keeping track of the most recent occurrences of P_1 and P_2.

$\Longrightarrow$ $O(|T| + |P_1| + |P_2| + occ)$ time algorithm.

Grammar compressed pattern matching

For **a single pattern** P, matching in a text T given as a grammar of size g, **[Ganardi & Gawrychowski, SODA'22]** showed that we can detect whether P occurs in T in $O(g + |P|)$ **time.**

Can we extend this result to two patterns consecutive (and reporting)? Yes!

Gawrychowski, Gourdel, Starikovskaya, Steiner (Unpublished)

Given T of length N as grammar of size g, and two string patterns P_1, P_2, we can report all consecutive occurrences of P_1, P_2 in T in $O(g + |P_1| + |P_2| + occ)$ time.

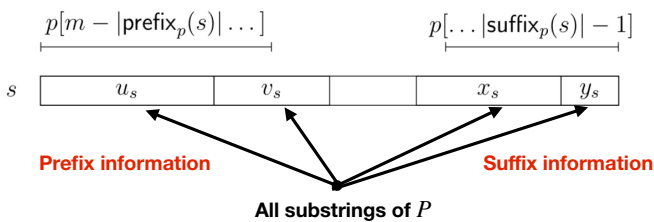
Boundary information

For a pattern P, the P-boundary information of a string S is substrings occurring both in P and S:

If S occurs in P, then the position where it occurs is the P-substring information.

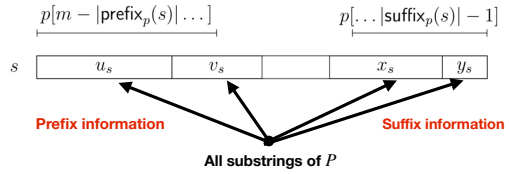
Else, let $\text{prefix}_P(S)$ be the **longest prefix** of S which is a suffix of P.

let $\text{suffix}_P(S)$ be the **longest suffix** of S which is a prefix of P.



P-boundary information of two strings S and T allows to efficiently report new occurrences of P appearing in ST.
+ the boundary information for ST can be computed quickly.

(Secondary) Boundary information



P-boundary information of two strings S and T allows to efficiently report new occurrences of P appearing in ST.
+ the boundary information for ST can be computed quickly.

For an SLP rule $A \rightarrow BC$, Compute bottom to top:

- P_1-boundary information and P_2-boundary information for $\overline{A}$ (from the boundary informations for $\overline{B}$ and $\overline{C}$).
- All crossing occurrences of P_1, P_2 in A.
- The rightmost occurrences of P_1, P_2 in $\overline{A}$.
- If the P_2-suffix information for $\overline{A}$ is (x_A, y_A): P_1-boundary information for x_A and y_A, all crossing occurrences of P_1 and leftmost rightmost.

Enough to detect pirmary co-occ in $O(g + |P_1| + |P_2|)$ time !

Summary

Complex matching: **Consecutive occurrences:** given two string patterns P_1, P_2, retrieve all pairs of consecutive occurrences of P_1, P_2 .

Sketch as input: Grammar compressed input (handled efficiently through boundary information)

Pattern matching: process P and T at the same time.

Gawrychowski, Gourdel, Starikovskaya, Steiner (Unpublished)

Given T of length N as grammar of size g, and two string patterns P_1, P_2, we can report all consecutive occurrences of P_1, P_2 in T in $O(g + |P_1| + |P_2| + occ)$ time.

Corollary: Gapped consecutive matching in $O(g + |P_1| + |P_2| + occ)$ time and Top-k closest occurrences $O(g + |P_1| + |P_2| + k)$ time.