Indexing Compressed Text: a Tale of Time and Space

Nicola Prezza, LUISS Guido Carli, Rome

18th Symposium on Experimental Algorithms, Catania, Italy, June 16-18, 2020

Introduction

In this talk I will present a brief history and state-of-the-art of the problem of computing over compressed data.

In this talk I will present a brief history and state-of-the-art of the problem of computing over compressed data.

We will look at solutions for a specific problem (text indexing). In general, the question of the field is:

In this talk I will present a brief history and state-of-the-art of the problem of computing over compressed data.

We will look at solutions for a specific problem (text indexing). In general, the question of the field is:

"*I have a really good compressor that compresses my data X into an archive C, with size$(C) \ll$ size(X).*

Can I perform computation directly over C, without decompressing it?"

Compressed text indexing

In general, the solution depends on the compressor C and on the problem (i.e. input and queries).

Compressed text indexing

In general, the solution depends on the compressor C and on the problem (i.e. input and queries).

In this talk, we will see solutions for different Cs and one particular problem:

Compressed text indexing

In general, the solution depends on the compressor C and on the problem (i.e. input and queries).

In this talk, we will see solutions for different Cs and one particular problem:

Definition (text indexing) Given a string $S \in \Sigma^n$, build a data structure $D(S)$ that answers the following queries:

Compressed text indexing

In general, the solution depends on the compressor C and on the problem (i.e. input and queries).

In this talk, we will see solutions for different Cs and one particular problem:

Definition (text indexing) Given a string $S \in \Sigma^n$, build a data structure $D(S)$ that answers the following queries:

- **Count** the number *occ* of occurrences of a string $P \in \Sigma^m$, $m \leq n$ in S

Compressed text indexing

In general, the solution depends on the compressor C and on the problem (i.e. input and queries).

In this talk, we will see solutions for different Cs and one particular problem:

Definition (text indexing) Given a string $S \in \Sigma^n$, build a data structure $D(S)$ that answers the following queries:

- **Count** the number *occ* of occurrences of a string $P \in \Sigma^m$, $m \leq n$ in S
- **Locate** the *occ* of occurrences of P in S

Compressed text indexing

In general, the solution depends on the compressor C and on the problem (i.e. input and queries).

In this talk, we will see solutions for different Cs and one particular problem:

Definition (text indexing) Given a string $S \in \Sigma^n$, build a data structure $D(S)$ that answers the following queries:

- **Count** the number *occ* of occurrences of a string $P \in \Sigma^m$, $m \leq n$ in S
- **Locate** the *occ* of occurrences of P in S
- **Extract** a text substring $S[i, \ldots, i + \ell - 1]$

Compressed text indexing

In general, the solution depends on the compressor C and on the problem (i.e. input and queries).

In this talk, we will see solutions for different Cs and one particular problem:

Definition (text indexing) Given a string $S \in \Sigma^n$, build a data structure $D(S)$ that answers the following queries:

- **Count** the number *occ* of occurrences of a string $P \in \Sigma^m$, $m \leq n$ in S
- **Locate** the *occ* of occurrences of P in S
- **Extract** a text substring $S[i, \ldots, i + \ell - 1]$

Additional constraint: $D(S)$ should take space proportional to C (compressed).

Compressed text indexing

Example

$$S = \begin{matrix} A & T & A & T & A & G & A & T & A \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{matrix}$$

- **Count(ATA)** $= 3$
- **Locate(ATA)** $= \{1, 3, 7\}$
- **Extract(4,7)** $=$ "$TAGA$"

Note: because of the **extract** query, $D(S)$ replaces S (we call it a *self-index*).

Entropy Compression

Zero-Order Empirical Entropy

At first, research focused on Shannon's measure of *text entropy*.

In order to do so, we first need to adapt the definition to the *empirical character frequencies* (we work on texts, not on character sources):

Zero-Order Empirical Entropy

At first, research focused on Shannon's measure of *text entropy*.

In order to do so, we first need to adapt the definition to the *empirical character frequencies* (we work on texts, not on character sources):

Definition (Zero-Order Empirical Entropy)

$$H_0(S) = \sum_{c \in \Sigma} \frac{occ_c}{n} \log_2 \frac{n}{occ_c}$$

where occ_c = number of occurrences of character c in S.

Zero-Order Empirical Entropy

At first, research focused on Shannon's measure of *text entropy*.

In order to do so, we first need to adapt the definition to the *empirical character frequencies* (we work on texts, not on character sources):

Definition (Zero-Order Empirical Entropy)

$$H_0(S) = \sum_{c \in \Sigma} \frac{occ_c}{n} \log_2 \frac{n}{occ_c}$$

where occ_c = number of occurrences of character c in S.

Thm. $nH_0(S)$ bits are needed to represent a text using any encoding of the alphabet's characters into binary codes that only depend on the character's frequency.

High-Order Empirical Entropy

A more powerful notion clusters symbols by context.

High-Order Empirical Entropy

A more powerful notion clusters symbols by context.

Let $S_C =$ sting obtained by concatenating all characters that follow substring C in S.

Example: in $S = AAATAAGCT$, $\quad S_{AA} = "ATG"$

High-Order Empirical Entropy

A more powerful notion clusters symbols by context.

Let S_C = sting obtained by concatenating all characters that follow substring C in S.

Example: in $S = AAATAAGCT$, $S_{AA} =$ "ATG"

Definition (High-Order Empirical Entropy*)

$$H_k = \sum_{C \in \Sigma^k} \frac{|S_C|}{n} \cdot H_0(S_C)$$

Intuition: weighted average of the contexts' zero-order entropies.

*From now on we will simply write H_k instead of $H_k(S)$

High-Order Empirical Entropy

Entropy compressors (e.g. Huffman, arithmetic) compress S into $nH_k + o(n \log \sigma)$ bits, for some $k \leq \log_\sigma n$ * ($\sigma = |\Sigma|$ = alphabet size)

On typical context-predictable texts, e.g. XML:

- nH_0 is about 65% of $n \log \sigma$.
- nH_5 is about 10% of $n \log \sigma$.

* We cannot do much better than that: Gagie [Inf. Proc. Letters, 2016] showed that for $k \geq \log_\sigma n$, no compressed representation can achieve a worst-case space bound of $\Theta(nH_k) + o(n \log \sigma)$

Goal: build a text index taking $O(nH_k) + o(n \log \sigma)$ bits of space and supporting fast queries.

Goal: build a text index taking $O(nH_k) + o(n \log \sigma)$ bits of space and supporting fast queries.

Classic solutions: **suffix trees, suffix arrays**. Fast, but use $O(n \log n)$ bits of space, which could be two orders of magnitude larger than nH_k.

Goal: build a text index taking $O(nH_k) + o(n \log \sigma)$ bits of space and supporting fast queries.

Classic solutions: **suffix trees, suffix arrays**. Fast, but use $O(n \log n)$ bits of space, which could be two orders of magnitude larger than nH_k.

Let's see (in 1 slide!) what is and how to compress a suffix array

Input $-terminated text ($\$ \prec_{lex} c$ for all $c \in \Sigma$)

S	$=$	A	T	A	T	A	G	A	T	$\$$
		1	2	3	4	5	6	7	8	9

Input $-terminated text ($\$ \prec_{lex} c$ for all $c \in \Sigma$)

S	$=$	A	T	A	T	A	G	A	T	$\$$
		1	2	3	4	5	6	7	8	9

Suffix Array: sort positions by lexicographic order of suffixes:

SA	$=$	9	5	7	3	1	6	8	4	2
		$\$$	A	A	A	A	G	T	T	T
			G	T	T	T	A	$\$$	A	A
			A	$\$$	A	A	T		G	T
			T		G	T	$\$$		A	A
			$\$$		A	A			T	G
					T	G			$\$$	A
					$\$$	A				T
						T				$\$$
						$\$$				

Note: occurrences of a pattern form a range: count/locate = binary search.

Input \$-terminated text (\$ $\prec_{lex}$ c for all $c \in \Sigma$)

S	=	A	T	A	T	A	G	A	T	\$
		1	2	3	4	5	6	7	8	9

ψ **Array**: $\psi[i] = SA^{-1}[SA[i] + 1]$ *

SA	=	9	5	7	3	1	6	8	4	2
ψ	=	5	6	7	8	9	3	1	2	4
		1	2	3	4	5	6	7	8	9

* **except** $\psi[1] = SA^{-1}[1]$

Input $-terminated text ($\$ \prec_{lex} c$ for all $c \in \Sigma$)

S	$=$	A	T	A	T	A	G	A	T	$\$$
		1	2	3	4	5	6	7	8	9

ψ **Array**: $\psi[i] = SA^{-1}[SA[i]+1]$ *

SA	$=$	9	5	7	3	1	6	8	4	2
ψ	$=$	5	6	7	8	9	3	1	2	4
		1	2	3	4	5	6	7	8	9

- Note: ψ is increasing by letter (color).

Input $-terminated text ($\$ \prec_{lex} c$ for all $c \in \Sigma$)

S	$=$	A	T	A	T	A	G	A	T	$\$$
		1	2	3	4	5	6	7	8	9

ψ **Array**: $\psi[i] = SA^{-1}[SA[i]+1]$ *

SA	$=$	9	5	7	3	1	6	8	4	2
ψ	$=$	5	6	7	8	9	3	1	2	4
		1	2	3	4	5	6	7	8	9

- Note: ψ is increasing by letter (color).
- Why? applying ψ = removing the first char from a suffix. Preserves relative ordering of suffixes starting with same letter

Input $-terminated text ($\$ \prec_{lex} c$ for all $c \in \Sigma$)

S	$=$	A	T	A	T	A	G	A	T	$\$$
		1	2	3	4	5	6	7	8	9

ψ **Array**: $\psi[i] = SA^{-1}[SA[i] + 1]$ *

SA	$=$	9	5	7	3	1	6	8	4	2
ψ	$=$	5	6	7	8	9	3	1	2	4
		1	2	3	4	5	6	7	8	9

- Note: ψ is increasing by letter (color).
- Why? applying ψ = removing the first char from a suffix. Preserves relative ordering of suffixes starting with same letter
- Store $\Delta[i] = \psi[i] - \psi[i-1]$ (delta-encoding): $nH_0 + O(n)$ bits, $O(1)$ random access.

Extract text using ψ

Let's see how to extract the suffix starting in position $SA[5]$.
We store: ψ and first letters (underlined). Space: $nH_0 + O(n)$ bits.

	1	2	3	4	5	6	7	8	9
ψ =	5	6	7	8	9	3	1	2	4
	$	A	A	A	A	G	T	T	T
		G	T	T	T	A	$	A	A
		A	$	A	A	T		G	T
		T		G	T	$		A	A
		$		A	A			T	G
				T	G			$	A
				$	A				T
					T				$
					$				

Extracted: A

Extract text using ψ

Let's see how to extract the suffix starting in position $SA[5]$.

We store: ψ and first letters (underlined). Space: $nH_0 + O(n)$ bits.

	1	2	3	4	5	6	7	8	9
$\psi =$	5	6	7	8	9	3	1	2	4
	$	A	A	A	A	G	T	T	T
		G	T	T	T	A	$	A	A
		A	$	A	A	T		G	T
		T		G	T	$		A	A
		$		A	A			T	G
				T	G			$	A
				$	A				T
					T				$
					$				

Extracted: AT

Extract text using ψ

Let's see how to extract the suffix starting in position $SA[5]$.
We store: ψ and first letters (underlined). Space: $nH_0 + O(n)$ bits.

	1	2	3	4	5	6	7	8	9
ψ =	5	6	7	8	9	3	1	2	4
	$	A	A	A	A	G	T	T	T
		G	T	T	T	A	$	A	A
		A	$	A	A	T		G	T
		T		G	T	$		A	A
		$		A	A			T	G
				T	G			$	A
				$	A				T
					T				$
					$				

Extracted: ATA

Extract text using ψ

Let's see how to extract the suffix starting in position $SA[5]$.
We store: ψ and first letters (underlined). Space: $nH_0 + O(n)$ bits.

	1	2	3	4	5	6	7	8	9
ψ =	5	6	7	8	9	3	1	2	4
	$	A	A	A	A	G	T	T	T
		G	T	T	T	A	$	A	A
		A	$	A	A	T		G	T
		T		G	T	$		A	A
		$		A	A			T	G
				T	G			$	A
				$	A				T
					T				$
					$				

Extracted: ATAT

The Compressed Suffix Array

The range of suffixes prefixed by a pattern P can be found with binary search using ψ.

The Compressed Suffix Array

The range of suffixes prefixed by a pattern P can be found with binary search using ψ.

By sampling the suffix array every $O(\log n)$ text positions, we obtain a **Compressed Suffix Array**.

The Compressed Suffix Array

Trade-offs (later slightly improved):

- **Space**: $nH_0 + O(n)$ bits.
- **Count**: $O(m \log n)$.
- **Locate**: $O((m + occ) \log n)$ (needs a sampling of SA)
- **Extract**: $O(\ell + \log n)$ (needs a sampling of SA^{-1})

First described in:

Grossi, Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In STOC 2000 (pp. 397-406).

High-Order Compression

We achieved nH_0. What about nH_k?

High-Order Compression

We achieved nH_0. What about nH_k?

We use an apparently different (but actually equivalent) idea: the Burrows-Wheeler Transform (BWT, Burrows, Wheeler, 1994)

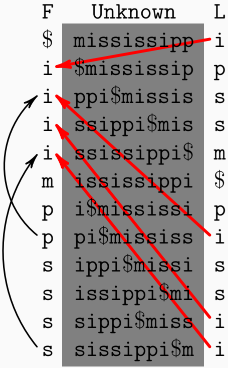
Burrows-Wheeler Transform

Sort all circular permutations of $S = mississippi\$$. BWT = last column.

F											**L**
$	m	i	s	s	i	s	s	i	p	p	i
i	$	m	i	s	s	i	s	s	i	p	p
i	p	p	i	$	m	i	s	s	i	s	s
i	s	s	i	p	p	i	$	m	i	s	s
i	s	s	i	s	s	i	p	p	i	$	m
m	i	s	s	i	s	s	i	p	p	i	$
p	i	$	m	i	s	s	i	s	s	i	p
p	p	i	$	m	i	s	s	i	s	s	i
s	i	p	p	i	$	m	i	s	s	i	s
s	i	s	s	i	p	p	i	$	m	i	s
s	s	i	p	p	i	$	m	i	s	s	i
s	s	i	s	s	i	p	p	i	$	m	i

Explicitly store only first and last columns.

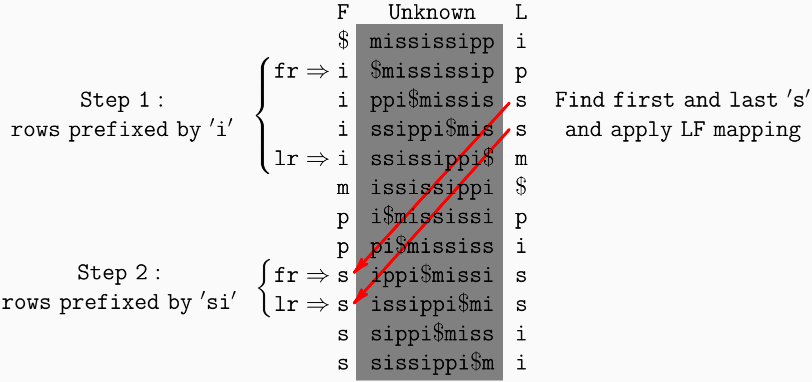
LF property. Let $c \in \Sigma$. Then, the i-th occurrence of c in L corresponds to the i-th occurrence of c in F (i.e. same position in T).



Red arrows: LF function (only character 'i' is shown)
Black arrows: implicit backward links (backward navigation of T)

Backward search

Backward search of the pattern 'si'



Burrows-Wheeler Transform

Finally, note: in BWT, characters are partitioned by context (example: $k = 2$)

F											L
$	m	i	s	s	i	s	s	i	p	p	i
i	$	m	i	s	s	i	s	s	i	p	p
i	p	p	i	$	m	i	s	s	i	s	s
i	s	s	i	p	p	i	$	m	i	s	s
i	s	s	i	s	s	i	p	p	i	$	m
m	i	s	s	i	s	s	i	p	p	i	$
p	i	$	m	i	s	s	i	s	s	i	p
p	p	i	$	m	i	s	s	i	s	s	i
s	i	p	p	i	$	m	i	s	s	i	s
s	i	s	s	i	p	p	i	$	m	i	s
s	s	i	p	p	i	$	m	i	s	s	i
s	s	i	s	s	i	p	p	i	$	m	i

We can compress each context independently using a zero-order compressor (e.g. Huffman) and obtain nH_k

The FM index

This structure is known as **FM-index**. Simplified trade-offs (later improved):

- **Space**: $nH_k + o(n \log \sigma)$ bits for $k = \alpha \log_\sigma n - 1$, $0 < \alpha < 1$.
- **Count**: $O(m \log \sigma)$.
- **Locate**: $O(m \log \sigma + occ \log^{1+\epsilon} n)$ (needs a sampling of SA)
- **Extract**: $O(\ell \log \sigma + \log^{1+\epsilon} n)$ (needs a sampling of SA^{-1})

First described (with slightly different trade-offs) in:

Ferragina, Manzini. Opportunistic data structures with applications. In FOCS 2000, Nov 12 (pp. 390-398).

The FM index

This structure is known as **FM-index**. Simplified trade-offs (later improved):

- **Space**: $nH_k + o(n \log \sigma)$ bits for $k = \alpha \log_\sigma n - 1$, $0 < \alpha < 1$.
- **Count**: $O(m \log \sigma)$.
- **Locate**: $O(m \log \sigma + occ \log^{1+\epsilon} n)$ (needs a sampling of SA)
- **Extract**: $O(\ell \log \sigma + \log^{1+\epsilon} n)$ (needs a sampling of SA^{-1})

First described (with slightly different trade-offs) in:

Ferragina, Manzini. Opportunistic data structures with applications. In FOCS 2000, Nov 12 (pp. 390-398).

Huge impact in medicine and bioinformatics: if you get your own genome sequenced, it will be analyzed using software based on the FM-index.

New data

The *compressed indexing* revolution happened in the early 2000s.

New data

The *compressed indexing* revolution happened in the early 2000s.

Then, **the data** changed!

New data

The *compressed indexing* revolution happened in the early 2000s.

Then, **the data** changed!

The last decade has been characterized by an explosion in the production of **highly repetitive massive data**

New data

The *compressed indexing* revolution happened in the early 2000s.

Then, **the data** changed!

The last decade has been characterized by an explosion in the production of **highly repetitive massive data**

- DNA repositories (1000genomes project, sequencing,...)

New data

The *compressed indexing* revolution happened in the early 2000s.

Then, **the data** changed!

The last decade has been characterized by an explosion in the production of **highly repetitive massive data**

- DNA repositories (1000genomes project, sequencing,...)
- Versioned repositories (wikipedia, github, ...)

Entropy is no longer a good model

Limitations of entropy became apparent: being memory-less, entropy is **insensitive to long repetitions** (remember: context length k is small!).

- $H_0(\texttt{banana}) \approx 1.45$

Entropy is no longer a good model

Limitations of entropy became apparent: being memory-less, entropy is **insensitive to long repetitions** (remember: context length k is small!).

- $H_0(\texttt{banana}) \approx 1.45$
- $H_0(\texttt{bananabanana}) \approx 1.45$

Entropy is no longer a good model

Limitations of entropy became apparent: being memory-less, entropy is **insensitive to long repetitions** (remember: context length k is small!).

- $H_0(\texttt{banana}) \approx 1.45$
- $H_0(\texttt{bananabanana}) \approx 1.45$
- $H_0(\texttt{bananabananabanana}) \approx 1.45$
- ...

Beating entropy

As a result, $S^3 = \texttt{bananabananabanana}$ compresses to $|S^3|H(S^3) = \mathbf{3 \cdot |S|H(S)}$ **bits** ...

Beating entropy

As a result, $S^3 = \texttt{bananabananabanana}$ compresses to $|S^3|H(S^3) = \mathbf{3} \cdot |\mathbf{S}|\mathbf{H}(\mathbf{S})$ **bits** ...

Can you come up with a better compressor?

Beating entropy

As a result, $S^3 = \texttt{bananabananabanana}$ compresses to $|S^3|H(S^3) = \mathbf{3 \cdot |S|H(S)}$ **bits** ...

Can you come up with a better compressor?

`compress` (

) =

$\times\ 5$

Beating entropy

As a result, $S^3 =$ `bananabananabanana` compresses to
$|S^3|H(S^3) = \mathbf{3} \cdot |\mathbf{S}|\mathbf{H}(\mathbf{S})$ **bits** …

Can you come up with a better compressor?

`compress` () =  $\times 5$

$$|S|H(S) + \mathcal{O}(\log t) \ll t \cdot |S|H(S) \text{ bits.}$$

Dictionary Compression

Ideal compressor: Kolmogorov complexity.

Ideal compressor: Kolmogorov complexity. Non computable/approximable!

Ideal compressor: Kolmogorov complexity. Non computable/approximable!

$\Rightarrow$ We need to fix a text model: exact repetitions

Ideal compressor: Kolmogorov complexity. Non computable/approximable!

$\Rightarrow$ We need to fix a text model: exact repetitions

A different generation of compressors comes at rescue: **Dictionary compressors**

General idea:

- Break S into substrings belonging to some dictionary D
- Represent S as pointers to D
- Usually, D is the set of substrings of S (self-referential compression)

LZ77 (Lempel-Ziv, 1977) — 7-zip, winzip

- LZ77 = Greedy partition of text into shortest factors not appearing before: `a|n|na|and|nan|ab|anan|anas|andb|ananas`

LZ77 (Lempel-Ziv, 1977) — 7-zip, winzip

- LZ77 = Greedy partition of text into shortest factors not appearing before: `a|n|na|and|nan|ab|anan|anas|andb|ananas`
- To encode each phrase: just a pointer back, phrase length, and 1 character: $|LZ77| = \mathcal{O}(\#\ of\ \ phrases)$

LZ77 (Lempel-Ziv, 1977) — 7-zip, winzip

- LZ77 = Greedy partition of text into shortest factors not appearing before: `a|n|na|and|nan|ab|anan|anas|andb|ananas`
- To encode each phrase: just a pointer back, phrase length, and 1 character: $|LZ77| = \mathcal{O}(\#\ of\ phrases)$
- Compresses **orders of magnitude better** than entropy on repetitive texts

Run-length BWT — bzip2

Input: $S = \text{BANANA}$

1. Build the matrix of all circular permutations

B	A	N	A	N	A	$
A	N	A	N	A	$	B
N	A	N	A	$	B	A
A	N	A	$	B	A	N
N	A	$	B	A	N	A
A	$	B	A	N	A	N
$	B	A	N	A	N	A

Run-Length Burrows-Wheeler Transform (RLBWT)

Run-length BWT — bzip2

Input: $S =$ BANANA

1. Build the matrix of all circular permutations

B	A	N	A	N	A	$
A	N	A	N	A	$	B
N	A	N	A	$	B	A
A	N	A	$	B	A	N
N	A	$	B	A	N	A
A	$	B	A	N	A	N
$	B	A	N	A	N	A

2. Sort the rows. BWT = last column.

						BWT
$	B	A	N	A	N	A
A	$	B	A	N	A	N
A	N	A	$	B	A	N
A	N	A	N	A	$	B
B	A	N	A	N	A	$
N	A	$	B	A	N	A
N	A	N	A	$	B	A

Run-Length Burrows-Wheeler Transform (RLBWT)

Run-length BWT — bzip2

Input: $S = \text{BANANA}$

1. Build the matrix of all circular permutations

B	A	N	A	N	A	$
A	N	A	N	A	$	B
N	A	N	A	$	B	A
A	N	A	$	B	A	N
N	A	$	B	A	N	A
A	$	B	A	N	A	N
$	B	A	N	A	N	A

2. Sort the rows. BWT = last column.

						BWT
$	B	A	N	A	N	A
A	$	B	A	N	A	N
A	N	A	$	B	A	N
A	N	A	N	A	$	B
B	A	N	A	N	A	$
N	A	$	B	A	N	A
N	A	N	A	$	B	A

3. Apply run-length compression to $BWT = \text{ANNB\$AA}$

Run-length BWT — bzip2

Input: S = BANANA

1. Build the matrix of all circular permutations

B	A	N	A	N	A	$
A	N	A	N	A	$	B
N	A	N	A	$	B	A
A	N	A	$	B	A	N
N	A	$	B	A	N	A
A	$	B	A	N	A	N
$	B	A	N	A	N	A

2. Sort the rows. BWT = last column.

						BWT
$	B	A	N	A	N	A
A	$	B	A	N	A	N
A	N	A	$	B	A	N
A	N	A	N	A	$	B
B	A	N	A	N	A	$
N	A	$	B	A	N	A
N	A	N	A	$	B	A

3. Apply run-length compression to *BWT* = ANNB$AA

Output: RLBWT = (1,A), (2,N), (1,B), (1,$), (2,A)

Highly repetitive text collections

How do these compressors perform in practice?

Real-case example

- All revisions of `en.wikipedia.org/wiki/Albert_Einstein`

How do these compressors perform in practice?

Real-case example

- All revisions of `en.wikipedia.org/wiki/Albert_Einstein`
- Uncompressed: 456 MB

How do these compressors perform in practice?

Real-case example

- All revisions of `en.wikipedia.org/wiki/Albert_Einstein`
- Uncompressed: 456 MB
- $nH_5 \approx 110MB$. **4x compression rate**.

Highly repetitive text collections

How do these compressors perform in practice?

Real-case example

- All revisions of `en.wikipedia.org/wiki/Albert_Einstein`
- Uncompressed: 456 MB
- $nH_5 \approx 110MB$. **4x compression rate**.
- $|RLBWT(T)| \approx 544KB$. **840x compression rate**.

How do these compressors perform in practice?

Real-case example

- All revisions of `en.wikipedia.org/wiki/Albert_Einstein`
- Uncompressed: 456 MB
- $nH_5 \approx 110MB$. **4x compression rate**.
- $|RLBWT(T)| \approx 544KB$. **840x compression rate**.
- $|LZ77(T)| \approx 310KB$. **1400x compression rate**.

Dictionary compressors

Known dictionary compressors (compressed size between parentheses):

1. **RLBWT** (r)
2. **LZ77** (z)
3. **macro schemes** (b) = bidirectional LZ77 [Storer, Szymanski '78]
4. **SLP**s (g) = context-free grammar generating S [Kieffer, Yang '00]
5. **RLSLP**s (g_{rl}) = SLPs with run-length rules $Z \rightarrow A^{\ell}$ [Nishimoto et al. '16]
6. **collage systems** (c) = RLSLPs with substring operator [Kida et al. '03]
7. **word graphs** (e) = automata accepting S's substrings [Blumer et al. '87]

(3-6) NP-hard to optimize

Note the zoo of compressibility measures (we'll come back to this later)

Can we build compressed indexes taking $|RLBWT|$ or $|LZ77|$ space?

Can we build compressed indexes taking $|RLBWT|$ or $|LZ77|$ space?

Notation:

- r = number of equal-letter runs in the BWT

Can we build compressed indexes taking $|RLBWT|$ or $|LZ77|$ space?

Notation:

- r = number of equal-letter runs in the BWT
- z = number of phrases in the Lempel-Ziv parse

Can we build compressed indexes taking $|RLBWT|$ or $|LZ77|$ space?

Notation:

- r = number of equal-letter runs in the BWT
- z = number of phrases in the Lempel-Ziv parse

Note: while it can be proven that z, r are related to nH_k, we don't actually want to do that: we will measure space complexity as a function of z, r.

Given the success of Compressed Suffix Arrays, the first natural try has been to run-length compress them.

The run-length FM index (RLFM-index)

2010: the **Run-Length CSA (RLCSA)**

name	space (words/bits)	Count	Locate	Extract
suffix tree ('73)	$\mathcal{O}(n)$ words	$\mathcal{O}(m)$	$\mathcal{O}(m + occ)$	$\mathcal{O}(\ell)$
suffix array ('93)	$2n$ words + text	$\mathcal{O}(m)$	$\mathcal{O}(m + occ)$	$\mathcal{O}(\ell)$
CSA ('00)	$nH_0 + O(n)$ bits	$\tilde{\mathcal{O}}(m)$	$\tilde{\mathcal{O}}(m + occ)$	$\tilde{\mathcal{O}}(\ell)$
FM-index ('00)	$nH_k + o(n \log \sigma)$ bits	$\tilde{\mathcal{O}}(m)$	$\tilde{\mathcal{O}}(m + occ)$	$\tilde{\mathcal{O}}(\ell)$
RLCSA ('10)	$\mathcal{O}(r + n/d)$ words	$\tilde{\mathcal{O}}(m)$	$\tilde{\mathcal{O}}(m + occ \cdot d)$	$\tilde{\mathcal{O}}(\ell + d)$

Mäkinen, Navarro, Sirén, and Välimäki. Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology, 2010

The run-length FM index (RLFM-index)

2010: the **Run-Length CSA (RLCSA)**

name	space (words/bits)	Count	Locate	Extract
suffix tree ('73)	$\mathcal{O}(n)$ words	$\mathcal{O}(m)$	$\mathcal{O}(m + occ)$	$\mathcal{O}(\ell)$
suffix array ('93)	$2n$ words + text	$\mathcal{O}(m)$	$\mathcal{O}(m + occ)$	$\mathcal{O}(\ell)$
CSA ('00)	$nH_0 + O(n)$ bits	$\tilde{\mathcal{O}}(m)$	$\tilde{\mathcal{O}}(m + occ)$	$\tilde{\mathcal{O}}(\ell)$
FM-index ('00)	$nH_k + o(n \log \sigma)$ bits	$\tilde{\mathcal{O}}(m)$	$\tilde{\mathcal{O}}(m + occ)$	$\tilde{\mathcal{O}}(\ell)$
RLCSA ('10)	$\mathcal{O}(r + n/d)$ words	$\tilde{\mathcal{O}}(m)$	$\tilde{\mathcal{O}}(m + occ \cdot d)$	$\tilde{\mathcal{O}}(\ell + d)$

Mäkinen, Navarro, Sirén, and Välimäki. Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology, 2010

Issue: The trade-off d (sampling rate of the suffix array) makes the index impractical on highly-repetitive texts (where $r \ll n$)

LZ indexing

What about Lempel-Ziv indexing?

index	compression	space (words)	locate time
KU-LZI[1]	LZ78	$\mathcal{O}(z) + n$	$\tilde{\mathcal{O}}(m^2 + occ)$
NAV-LZI[2]	LZ78	$\mathcal{O}(z)$	$\tilde{\mathcal{O}}(m^3 + occ)$
KN-LZI[3]	LZ77	$\mathcal{O}(z)$	$\tilde{\mathcal{O}}(m^2 h + occ)$

$h \leq n$ is the parse height. In practice small, but worst-case $h = \Theta(n)$

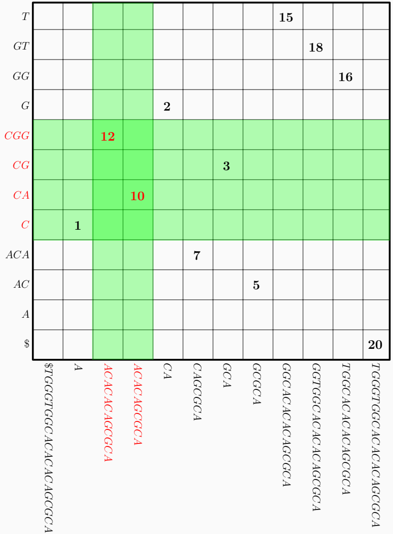
[1] Kärkkäinen, Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. InProc. 3rd South American Workshop on String Processing (WSP'96)

[2] Navarro. Indexing text using the Ziv-Lempel trie. Journal of Discrete Algorithms. 2004 Mar 1;2(1):87-114.

[3] Kreft, Navarro. On compressing and indexing repetitive sequences. Theoretical Computer Science. 2013 Apr 29;483:115-33.

Example: search splitted-pattern $\overleftarrow{CA}|\overrightarrow{C}$ (to find *all* splitted occurrences, we have to try all possible splits)

```
       0   1   2   3 4   5 6   7 8 9   10 11   12 13 14   15   16 17   18 19   20
LZ78 = A | C | G | C G | A C | A C A | C  A | C  G  G | T | G  G | G  T | $
```



Problems:

- Locate time quadratic in m
- These index cannot count (without locating)!

The problem has recently (2018) been solved going back to Run-Length CSAs:

The problem has recently (2018) been solved going back to Run-Length CSAs:

Theorem [1] Let $SA[l, \ldots, r]$ be the suffix array range of a pattern P. We can sample r positions of the suffix array (at BWT run-borders) such that:

[1] Gagie, Navarro, P. Optimal-time text indexing in BWT-runs bounded space. In SODA 2018.

[2] Gagie, Navarro, and P., 2020. Fully-Functional Suffix Trees and Optimal Text Searching in BWT-runs Bounded Space. Journal of the ACM

The problem has recently (2018) been solved going back to Run-Length CSAs:

Theorem [1] Let $SA[l, \ldots, r]$ be the suffix array range of a pattern P. We can sample r positions of the suffix array (at BWT run-borders) such that:

1. We can return $SA[l]$ in $O(m \log \log n)$ time

[1] Gagie, Navarro, P. Optimal-time text indexing in BWT-runs bounded space. In SODA 2018.

[2] Gagie, Navarro, and P., 2020. Fully-Functional Suffix Trees and Optimal Text Searching in BWT-runs Bounded Space. Journal of the ACM

The problem has recently (2018) been solved going back to Run-Length CSAs:

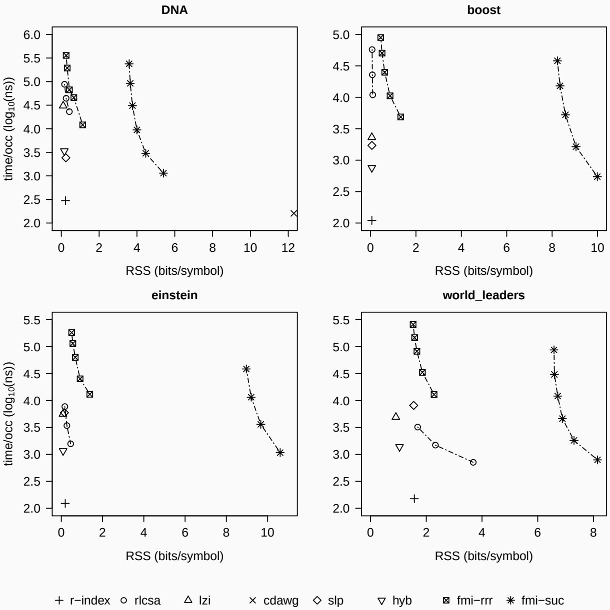
Theorem [1] Let $SA[l, \ldots, r]$ be the suffix array range of a pattern P. We can sample r positions of the suffix array (at BWT run-borders) such that:

1. We can return $SA[l]$ in $O(m \log\log n)$ time
2. Given $SA[i]$, we can compute $SA[i+1]$ in $O(\log\log n)$ time.

[1] Gagie, Navarro, P. Optimal-time text indexing in BWT-runs bounded space. In SODA 2018.

[2] Gagie, Navarro, and P., 2020. Fully-Functional Suffix Trees and Optimal Text Searching in BWT-runs Bounded Space. Journal of the ACM

smaller, orders of magnitude faster (`r-index`): the right tool to index thousands of genomes!



Exciting results:

- Index size for one human chromosome: 250 MB. 35 bps (bits per symbol).
- Index size for 1000 human chromosomes: 550 MB. **0.08 bps**
- **Faster** than the FM-index.

Up-to-date history of compressed suffix arrays:

name	space (words/bits)	Count	Locate	Extract
suffix tree ('73)	$\mathcal{O}(n)$ words	$\mathcal{O}(m)$	$\mathcal{O}(m + occ)$	$\mathcal{O}(\ell)$
suffix array ('93)	$2n$ words + text	$\mathcal{O}(m)$	$\mathcal{O}(m + occ)$	$\mathcal{O}(\ell)$
CSA ('00)	$nH_0 + O(n)$ bits	$\tilde{\mathcal{O}}(m)$	$\tilde{\mathcal{O}}(m + occ)$	$\tilde{\mathcal{O}}(\ell)$
FM-index ('00)	$nH_k + o(n \log \sigma)$ bits	$\tilde{\mathcal{O}}(m)$	$\tilde{\mathcal{O}}(m + occ)$	$\tilde{\mathcal{O}}(\ell)$
RLCSA ('10)	$\mathcal{O}(r + n/d)$ words	$\tilde{\mathcal{O}}(m)$	$\tilde{\mathcal{O}}(m + occ \cdot d)$	$\tilde{\mathcal{O}}(\ell + d)$
r-index [1,2] ('18)	$\mathcal{O}(r)$ words	$\tilde{\mathcal{O}}(m)$	$\tilde{\mathcal{O}}(m + occ)$	$\mathcal{O}(\ell + \log(n/r))^*$

[1] Gagie, Navarro, P. Optimal-time text indexing in BWT-runs bounded space. In SODA 2018.

[2] Gagie, Navarro, and P., 2020. Fully-Functional Suffix Trees and Optimal Text Searching in BWT-runs Bounded Space. Journal of the ACM

* only in space $O(r \log(n/r))$

Current directions

What next?

What next?

- Put some order in the zoo of complexity measures:
    - A definitive measure of "repetitiveness"
    - Relations between existing complexity measures

What next?

- Put some order in the zoo of complexity measures:
  - A definitive measure of "repetitiveness"
  - Relations between existing complexity measures
- Universal (compressor-independent) data structures

What next?

- Put some order in the zoo of complexity measures:
    - A definitive measure of "repetitiveness"
    - Relations between existing complexity measures
- Universal (compressor-independent) data structures
- Generalizations: indexing labeled graphs/regular languages

Universal Compression

String Attractors

String attractors [1]: a tentative to describe all complexity measures under the same framework. Observation:

- A repetitive string S has a small set of distinct substrings $\mathcal{Q} = \{S[i..j]\}$
- What if we fix a set of positions $\Gamma \subseteq [1..|S|]$ such that every $s \in \mathcal{Q}$ appears in S crossing some position of Γ?

[1] Kempa, P. At the roots of dictionary compression: String attractors. In STOC 2018.

String Attractors

String attractors [1]: a tentative to describe all complexity measures under the same framework. Observation:

- A repetitive string S has a small set of distinct substrings $\mathcal{Q} = \{S[i..j]\}$
- What if we fix a set of positions $\Gamma \subseteq [1..|S|]$ such that every $s \in \mathcal{Q}$ appears in S crossing some position of Γ?

We call Γ "**string attractor**". Intuition: few distinct substrings $\Rightarrow$ small Γ.

[1] Kempa, P. At the roots of dictionary compression: String attractors. In STOC 2018.

Example

$$S = \text{CDA}\underline{\text{B}}\text{CC}\underline{\text{D}}\text{ABC}\underline{\text{CA}} \qquad \Gamma = \{4, 7, 11, 12\}$$

in this case, Γ is also the *smallest* attractor ... why?

String Attractors

Main results:

- **Reductions** (universal: work for LZ77, RLBWT, grammars,...) [1]:
  - $|\Gamma| \le |\textit{dictionary compressors}| \le O(|\Gamma| \texttt{polylog}\ n)$

[1] Kempa and P. At the Roots of Dictionary Compression: String Attractors. STOC'18.

String Attractors

Main results:

- **Reductions** (universal: work for LZ77, RLBWT, grammars,...) [1]:
  - $|\Gamma| \leq |\textit{dictionary compressors}| \leq O(|\Gamma| \texttt{polylog}\ n)$
- Finding the smallest Γ is **NP-complete** and **APX-hard** [1]

[1] Kempa and P. At the Roots of Dictionary Compression: String Attractors. STOC'18.

String Attractors

Main results:

- **Reductions** (universal: work for LZ77, RLBWT, grammars,...) [1]:
  - $|\Gamma| \leq |\textit{dictionary compressors}| \leq O(|\Gamma| \texttt{polylog}\ n)$
- Finding the smallest Γ is **NP-complete** and **APX-hard** [1]
- **Optimal** universal **data structures** of size $\tilde{\mathcal{O}}(|\Gamma|)$ [1,2,4,5]

[1] Kempa and P. At the Roots of Dictionary Compression: String Attractors. STOC'18.
[2] Navarro and P. Universal Compressed Text Indexing. TCS'18.
[3] Kempa, Policriti, P., Rotenberg. String Attractors: Verification and Optimization. ESA'18.
[4] P. Optimal Rank and Select Queries on Dictionary-Compressed Text. CPM'19.
[5] Christiansen, Berggren Ettienne, Kociumaka, Navarro, P. Optimal-Time Dictionary-Compressed Indexes. arXiv preprint arXiv:1811.12779. 2018.

String Attractors

Main results:

- **Reductions** (universal: work for LZ77, RLBWT, grammars,...) [1]:
  - $|\Gamma| \leq |\textit{dictionary compressors}| \leq O(|\Gamma| \texttt{polylog}\ n)$
- Finding the smallest Γ is **NP-complete** and **APX-hard** [1]
- **Optimal** universal **data structures** of size $\tilde{\mathcal{O}}(|\Gamma|)$ [1,2,4,5]
- FPT algorithms + check if Γ is a valid attractor in linear time [3]

[1] Kempa and P. At the Roots of Dictionary Compression: String Attractors. STOC'18.
[2] Navarro and P. Universal Compressed Text Indexing. TCS'18.
[3] Kempa, Policriti, P., Rotenberg. String Attractors: Verification and Optimization. ESA'18.
[4] P. Optimal Rank and Select Queries on Dictionary-Compressed Text. CPM'19.
[5] Christiansen, Berggren Ettienne, Kociumaka, Navarro, P. Optimal-Time Dictionary-Compressed Indexes. arXiv preprint arXiv:1811.12779. 2018.

Indexing Graphs

Indexing graphs

Recently, the concept of prefix-sorting has been extended to graphs:

Wheeler graph [1]: an edge-labeled graph whose nodes can be prefix-sorted

[1] Gagie, Manzini, Sirén. Wheeler graphs: A framework for BWT-based data structures. TCS'17.

Indexing graphs

Recently, the concept of prefix-sorting has been extended to graphs:

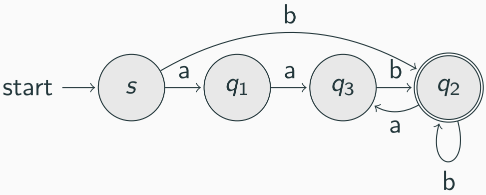
Wheeler graph [1]: an edge-labeled graph whose nodes can be prefix-sorted

FM-indexes + Wheeler Graphs = **path queries**: find nodes reachable (from any node) by a path labeled $w \in \Sigma^*$

[1] Gagie, Manzini, Sirén. Wheeler graphs: A framework for BWT-based data structures. TCS'17.

$\mathcal{L} = (\epsilon|aa)b(ab|b)^*$

Sorted Wheeler automaton:



Note: paths lead to ranges of states (e.g. $a \rightarrow [q_1, q_3]$).

Indexing graphs

Not all graphs are Wheeler, and they are hard to recognize! Main results:

Indexing graphs

Not all graphs are Wheeler, and they are hard to recognize! Main results:

- **Hardness results [1]**
    - Recognizinig/sorting Wheeler NFAs (WNFAs) is NP-complete
    - Remove min number of edges to obtain a W.G.: APX-complete

[1] Gibney, Thankachan. On the Hardness and Inapproximability of Recognizing Wheeler Graphs. ESA'19.

Indexing graphs

Not all graphs are Wheeler, and they are hard to recognize! Main results:

- **Hardness results [1]**
  - Recognizinig/sorting Wheeler NFAs (WNFAs) is NP-complete
  - Remove min number of edges to obtain a W.G.: APX-complete
- **Positive results: Indexing regular languages [2]**
  - $\mathit{WNFA} \overset{\mathit{powerset}}{\rightarrow} \mathit{WDFA}$ with linear blow-up
  - Recognizing/sorting WDFAs in linear time
  - WDFA minimization in $O(n \log n)$ time
  - Any acyclic DFA $\rightarrow$ smallest WDFA in almost-optimal time

[1] Gibney, Thankachan. On the Hardness and Inapproximability of Recognizing Wheeler Graphs. ESA'19.
[2] Alanko, D'Agostino, Policriti, and P. Regular Languages meet Prefix Sorting. SODA'20.

Future Challenges

Future Challenges

What next?

Future Challenges

What next?

- Index compressed graphs

Future Challenges

What next?

- Index compressed graphs
- Index super-classes of the Wheeler languages

Future Challenges

What next?

- Index compressed graphs
- Index super-classes of the Wheeler languages
- Better measures of repetitiveness

Future Challenges

What next?

- Index compressed graphs
- Index super-classes of the Wheeler languages
- Better measures of repetitiveness
- Practical compressed indexes (possibly dynamic)

Thank you for your attention! questions?