An LMS-based Grammar Self-Index with Local Consistency Properties

Diego Díaz-Domínguez[1,2] Gonzalo Navarro[1] Alejandro Pacheco[1]

[1]University of Chile

[2]University of Helsinki

October 6, 2021

The Grammar Self-Index

A **grammar self-index** of a string $S[1, n]$ (Claude et al. 2012, 2020) is constructed on top of a grammar $\mathcal{G}$ that only produces $S[1, n]$.

The Grammar Self-Index

A **grammar self-index** of a string $S[1, n]$ (Claude et al. 2012, 2020) is constructed on top of a grammar $\mathcal{G}$ that only produces $S[1, n]$.

If S highly-repetitive, then the size of $\mathcal{G}$ is small, meaning that the self-index is also small.

The Grammar Self-Index

A **grammar self-index** of a string $S[1, n]$ (Claude et al. 2012, 2020) is constructed on top of a grammar $\mathcal{G}$ that only produces $S[1, n]$.

If S highly-repetitive, then the size of $\mathcal{G}$ is small, meaning that the self-index is also small.

Let us denote the size of $\mathcal{G}$ as g and the number of nonterminals as r.

The Grammar Self-Index

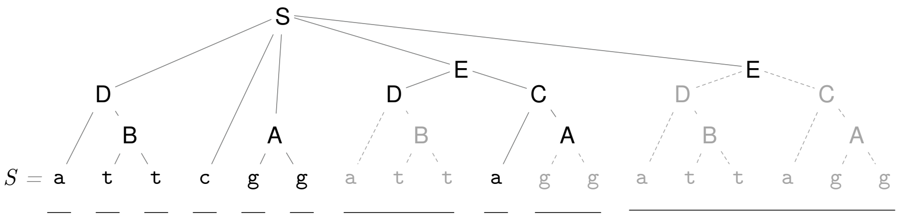
A **grammar self-index** of a string $S[1, n]$ (Claude et al. 2012, 2020) is constructed on top of a grammar $\mathcal{G}$ that only produces $S[1, n]$.

If S highly-repetitive, then the size of $\mathcal{G}$ is small, meaning that the self-index is also small.

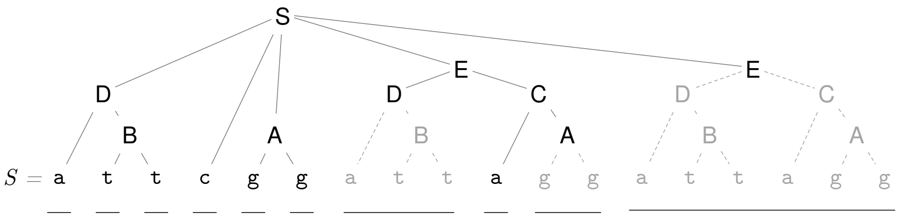
Let us denote the size of $\mathcal{G}$ as g and the number of nonterminals as r.

Claude et al. 2020 demonstrated that there is a self-index on top of $\mathcal{G}$ that requires $O(g \lg n + (2 + \epsilon) g \lg r)$ bits of space, where $0 < \epsilon \leq 1$ is a constant. This index can locate the *occ* the occurrences of a pattern $P[1, m]$ in $O((m^2 + occ) \lg g)$ time.

The Grammar Self-Index

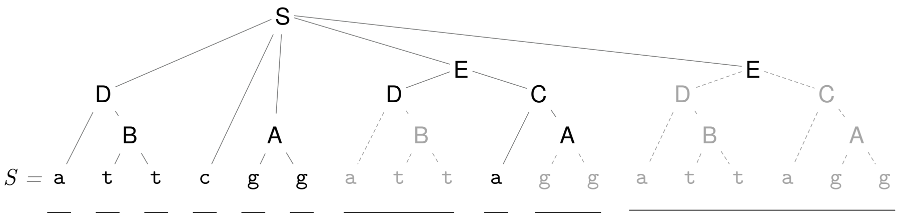


The Grammar Self-Index



A $\rightarrow$ g g	g|g
C $\rightarrow$ a A	a|gg
S $\rightarrow$ D c A E E	att|c|gg|attagg|attagg
E $\rightarrow$ D C	att|agg
B $\rightarrow$ t t	t|t
D $\rightarrow$ a B	a|tt

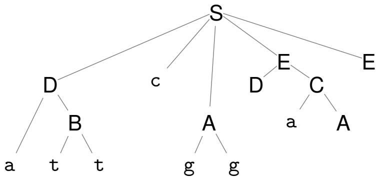
The Grammar Self-Index



A → g g	g|g	
C → a A	a|gg	$\mathcal{Y}$ $\mathcal{X}$
S → D c A E E	att|c|gg|attagg|attagg	att|cggattaggattagg
E → D C	att|agg	c|ggattaggattagg
B → t t	t|t	gg|attaggattagg
D → a B	a|tt	attagg|attagg

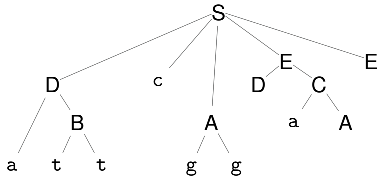
The Grammar Self-Index

	agg	attagg	attaggattagg	cggattaggattagg	g	gg	ggattaggattagg	t	tt
a						C			D
c							S		
g					A				
gg			S						
agg									
attcggattaggattagg									
attagg		S							
t								B	
tt									
att	E			S					



The Grammar Self-Index

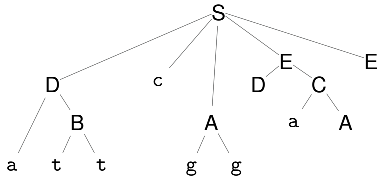
	agg	attagg	attaggattagg	cggattaggattagg	g	gg	ggattaggattagg	t	tt
a						C			D
c							S		
g					A				
gg			S						
agg									
attcggattaggattagg									
attagg		S							
t								B	
tt									
att	E			S					



P = a t t

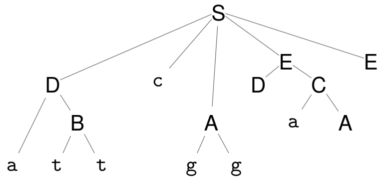
The Grammar Self-Index

	agg	attagg	attaggattagg	cggattaggattagg	g	gg	ggattaggattagg	t	tt
a						C			D
c							S		
g					A				
gg			S						
agg									
attcggattaggattagg									
attagg		S							
t								B	
tt									
att	E			S					



P = a|t t

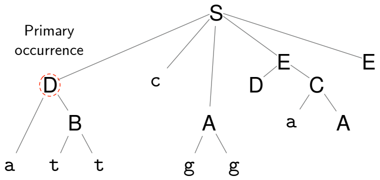
	agg	attagg	attaggattagg	cggattaggattagg	g	gg	ggattaggattagg	t	tt
a						C			D
c							S		
g					A				
gg			S						
agg									
attcggattaggattagg									
attagg		S							
t								B	
tt									
att	E			S					



P = a t t

The Grammar Self-Index

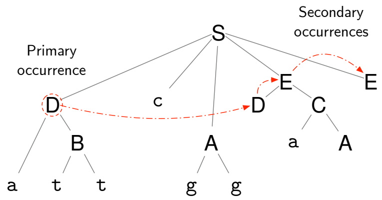
	agg	attagg	attaggattagg	cggattaggattagg	g	gg	ggattaggattagg	t	tt
a						C			D
c							S		
g					A				
gg			S						
agg									
attcggattaggattagg									
attagg		S							
t								B	
tt									
att	E			S					



P = a t t

The Grammar Self-Index

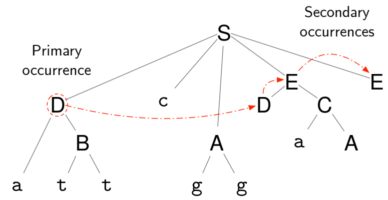
	agg	attagg	attaggattagg	cggattaggattagg	g	gg	ggattaggattagg	t	tt
a						C			D
c							S		
g					A				
gg			S						
agg									
attcggattaggattagg									
attagg		S							
t								B	
tt									
att	E			S					



P = a | t t

The Grammar Self-Index

	agg	attagg	attaggattagg	cggattaggattagg	g	gg	ggattaggattagg	t	tt
a						C			D
c							S		
g					A				
gg			S						
agg									
attcggattaggattagg									
attagg		S							
t								B	
tt									
att	E			S					



P = a | t t

Problem: We have to try out all the possible cuts of P

Locally consistent grammar self-index

Christiansen et al. 2021 reduced the time complexity for the *locate* operation in the grammar self-index to $O((m \log m + occ) \lg g)$ time.

Locally consistent grammar self-index

Christiansen et al. 2021 reduced the time complexity for the *locate* operation in the grammar self-index to $O((m \log m + occ) \lg g)$ time.

They achieve this time complexity by building the self-index with a **locally consistent** grammar.

Locally consistent grammar self-index

Christiansen et al. 2021 reduced the time complexity for the *locate* operation in the grammar self-index to $O((m \log m + occ) \lg g)$ time.

They achieve this time complexity by building the self-index with a **locally consistent** grammar.

- It is balanced.

Locally consistent grammar self-index

Christiansen et al. 2021 reduced the time complexity for the *locate* operation in the grammar self-index to $O((m \log m + occ) \lg g)$ time.

They achieve this time complexity by building the self-index with a **locally consistent** grammar.

- It is balanced.
- The occurrences of a pattern P are largely compressed in the same way.

Locally consistent grammar

The algorithm of Christiansen et al. 2020 has several rounds of parsing. In the first round $i = 1$, we set $S_i = S$ and apply the following procedure:

Locally consistent grammar

The algorithm of Christiansen et al. 2020 has several rounds of parsing. In the first round $i = 1$, we set $S_i = S$ and apply the following procedure:

1. Create a new string $\hat{S}_i$ by replacing the equal-symbol runs of S_i with new nonterminal symbols.

Locally consistent grammar

The algorithm of Christiansen et al. 2020 has several rounds of parsing. In the first round $i = 1$, we set $S_i = S$ and apply the following procedure:

1. Create a new string $\hat{S}_i$ by replacing the equal-symbol runs of S_i with new nonterminal symbols.
2. Select a random permutation π for the symbols in $\hat{S}_i$.

Locally consistent grammar

The algorithm of Christiansen et al. 2020 has several rounds of parsing. In the first round $i = 1$, we set $S_i = S$ and apply the following procedure:

1. Create a new string $\hat{S}_i$ by replacing the equal-symbol runs of S_i with new nonterminal symbols.
2. Select a random permutation π for the symbols in $\hat{S}_i$.
3. Partition $\hat{S}_i$ using π.

Locally consistent grammar

The algorithm of Christiansen et al. 2020 has several rounds of parsing. In the first round $i = 1$, we set $S_i = S$ and apply the following procedure:

1. Create a new string $\hat{S}_i$ by replacing the equal-symbol runs of S_i with new nonterminal symbols.
2. Select a random permutation π for the symbols in $\hat{S}_i$.
3. Partition $\hat{S}_i$ using π.
   - A local minima is a position $\hat{S}_i[a]$ such that $\pi(\hat{S}_i[a-1]) > \pi(\hat{S}_i[a]) < \pi(\hat{S}_i[a+1])$.

Locally consistent grammar

The algorithm of Christiansen et al. 2020 has several rounds of parsing. In the first round $i = 1$, we set $S_i = S$ and apply the following procedure:

1. Create a new string $\hat{S}_i$ by replacing the equal-symbol runs of S_i with new nonterminal symbols.
2. Select a random permutation π for the symbols in $\hat{S}_i$.
3. Partition $\hat{S}_i$ using π.
   - A local minima is a position $\hat{S}_i[a]$ such that $\pi(\hat{S}_i[a-1]) > \pi(\hat{S}_i[a]) < \pi(\hat{S}_i[a+1])$.
   - A phrase in the partition is every substring $\hat{S}_i[a, b]$ where $\hat{S}_i[a]$ and $\hat{S}_i[b+1]$ are local minima.

Locally consistent grammar

The algorithm of Christiansen et al. 2020 has several rounds of parsing. In the first round $i = 1$, we set $S_i = S$ and apply the following procedure:

1. Create a new string $\hat{S}_i$ by replacing the equal-symbol runs of S_i with new nonterminal symbols.
2. Select a random permutation π for the symbols in $\hat{S}_i$.
3. Partition $\hat{S}_i$ using π.
   - A local minima is a position $\hat{S}_i[a]$ such that $\pi(\hat{S}_i[a-1]) > \pi(\hat{S}_i[a]) < \pi(\hat{S}_i[a+1])$.
   - A phrase in the partition is every substring $\hat{S}_i[a, b]$ where $\hat{S}_i[a]$ and $\hat{S}_i[b+1]$ are local minima.
4. Create a new string $S_{i+1}[1, n_{i+1}]$, with $n_{i+1} < \lfloor n_i/2 \rfloor$, by replacing the phrases in $\hat{S}_i$ with new nonterminal symbols.

Locally consistent grammar

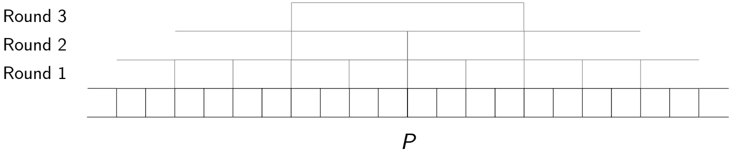
The algorithm of Christiansen et al. 2020 has several rounds of parsing. In the first round $i = 1$, we set $S_i = S$ and apply the following procedure:

1. Create a new string $\hat{S}_i$ by replacing the equal-symbol runs of S_i with new nonterminal symbols.
2. Select a random permutation π for the symbols in $\hat{S}_i$.
3. Partition $\hat{S}_i$ using π.
   - A local minima is a position $\hat{S}_i[a]$ such that $\pi(\hat{S}_i[a-1]) > \pi(\hat{S}_i[a]) < \pi(\hat{S}_i[a+1])$.
   - A phrase in the partition is every substring $\hat{S}_i[a, b]$ where $\hat{S}_i[a]$ and $\hat{S}_i[b+1]$ are local minima.
4. Create a new string $S_{i+1}[1, n_{i+1}]$, with $n_{i+1} < \lfloor n_i/2 \rfloor$, by replacing the phrases in $\hat{S}_i$ with new nonterminal symbols.
5. Repeat the same parsing algorithm with S_{i+1}

Locally consistent grammar and pattern matching

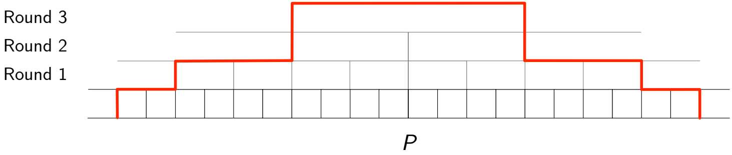
Let us assume the pattern P appears several times in S.

Let us assume the pattern P appears several times in S.



Locally consistent grammar and pattern matching

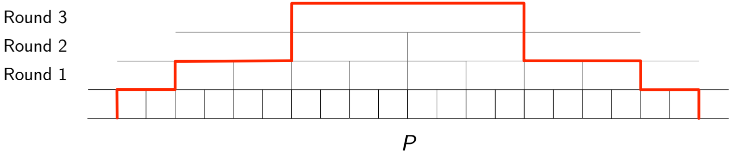
Let us assume the pattern P appears several times in S.



The phrases below the red line are always the same, regardless of the context of P in S.

Locally consistent grammar and pattern matching

Let us assume the pattern P appears several times in S.

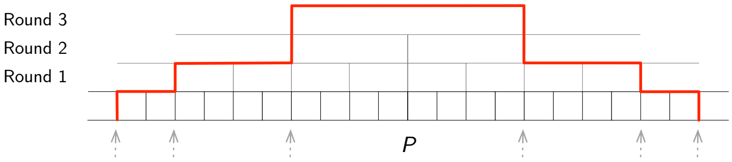


The phrases below the red line are always the same, regardless of the context of P in S.

Pattern matching: we parse P using the grammar algorithm, and try out in the grid of the grammar self-index the $O(\log m)$ cuts induced by the parsing.

Locally consistent grammar and pattern matching

Let us assume the pattern P appears several times in S.



The phrases below the red line are always the same, regardless of the context of P in S.

Pattern matching: we parse P using the grammar algorithm, and try out in the grid of the grammar self-index the $O(\log m)$ cuts induced by the parsing.

Locally consistent grammar and pattern matching

Disadvantages in the method of Christiansen et al. 2020:

- Building the grammar self-index requires storing the permutations to replicate the parsing procedure on any input pattern.

Locally consistent grammar and pattern matching

Disadvantages in the method of Christiansen et al. 2020:

- Building the grammar self-index requires storing the permutations to replicate the parsing procedure on any input pattern.
- The resulting grammar can be potentially large compared to other heuristics, like RePair.

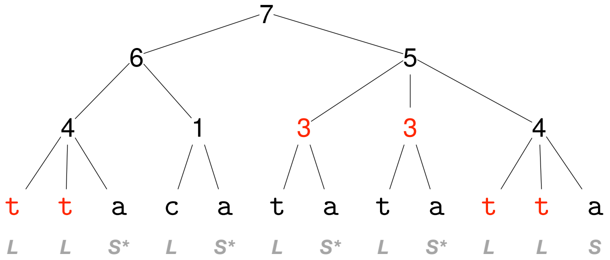
Locally consistent grammar and pattern matching

Disadvantages in the method of Christiansen et al. 2020:

- Building the grammar self-index requires storing the permutations to replicate the parsing procedure on any input pattern.
- The resulting grammar can be potentially large compared to other heuristics, like RePair.
- It increases the size of the grammar index considerably.

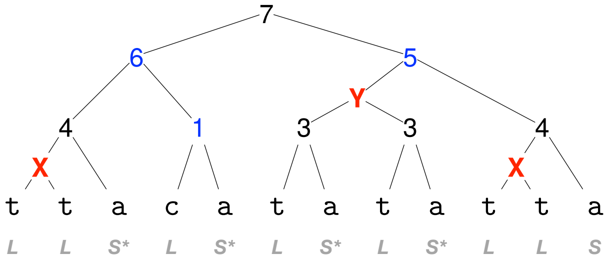
Our method

We build a locally consistent grammar using LMS parsing:



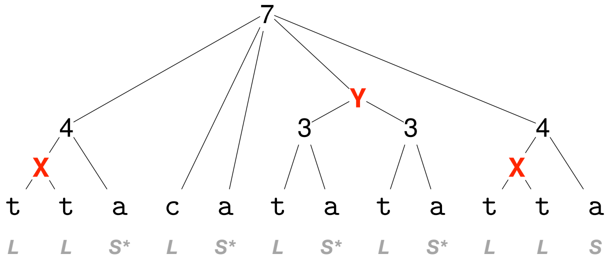
Our method

We build a locally consistent grammar using LMS parsing:



Our method

We build a locally consistent grammar using LMS parsing:



Experiments

- We implemented the grammar algorithm of Christiansen et al. 2020 and compared it with our LMS-based locally consistent algorithm.

Experiments

- We implemented the grammar algorithm of Christiansen et al. 2020 and compared it with our LMS-based locally consistent algorithm.
- We use the grammar resulted from our algorithm to build the self-index of Claude et al. 2020.

Experiments

- We implemented the grammar algorithm of Christiansen et al. 2020 and compared it with our LMS-based locally consistent algorithm.
- We use the grammar resulted from our algorithm to build the self-index of Claude et al. 2020.
  - We implemented two variations; `lms-ind` and `lms-ind-rrr`

Experiments

- We implemented the grammar algorithm of Christiansen et al. 2020 and compared it with our LMS-based locally consistent algorithm.
- We use the grammar resulted from our algorithm to build the self-index of Claude et al. 2020.
  - We implemented two variations; `lms-ind` and `lms-ind-rrr`
- We compared our version of the grammar self-index against the state-of-the-art dictionary-based self-indexes:

Experiments

- We implemented the grammar algorithm of Christiansen et al. 2020 and compared it with our LMS-based locally consistent algorithm.
- We use the grammar resulted from our algorithm to build the self-index of Claude et al. 2020.
  - We implemented two variations; `lms-ind` and `lms-ind-rrr`
- We compared our version of the grammar self-index against the state-of-the-art dictionary-based self-indexes:
  - `lz-ind`: a self-index based on the Lempel-Ziv parsing technique

Experiments

- We implemented the grammar algorithm of Christiansen et al. 2020 and compared it with our LMS-based locally consistent algorithm.
- We use the grammar resulted from our algorithm to build the self-index of Claude et al. 2020.
  - We implemented two variations; `lms-ind` and `lms-ind-rrr`
- We compared our version of the grammar self-index against the state-of-the-art dictionary-based self-indexes:
  - `lz-ind`: a self-index based on the Lempel-Ziv parsing technique
  - `slp-ind`: original version of the grammar index that works on straight-line program.

Experiments

- We implemented the grammar algorithm of Christiansen et al. 2020 and compared it with our LMS-based locally consistent algorithm.
- We use the grammar resulted from our algorithm to build the self-index of Claude et al. 2020.
  - We implemented two variations; `lms-ind` and `lms-ind-rrr`
- We compared our version of the grammar self-index against the state-of-the-art dictionary-based self-indexes:
  - `lz-ind`: a self-index based on the Lempel-Ziv parsing technique
  - `slp-ind`: original version of the grammar index that works on straight-line program.
  - `g-ind`: most recent version of the grammar index that works on any type of grammar.

Experiments

- We implemented the grammar algorithm of Christiansen et al. 2020 and compared it with our LMS-based locally consistent algorithm.
- We use the grammar resulted from our algorithm to build the self-index of Claude et al. 2020.
  - We implemented two variations; `lms-ind` and `lms-ind-rrr`
- We compared our version of the grammar self-index against the state-of-the-art dictionary-based self-indexes:
  - `lz-ind`: a self-index based on the Lempel-Ziv parsing technique
  - `slp-ind`: original version of the grammar index that works on straight-line program.
  - `g-ind`: most recent version of the grammar index that works on any type of grammar.
  - `r-ind`: the r-index.

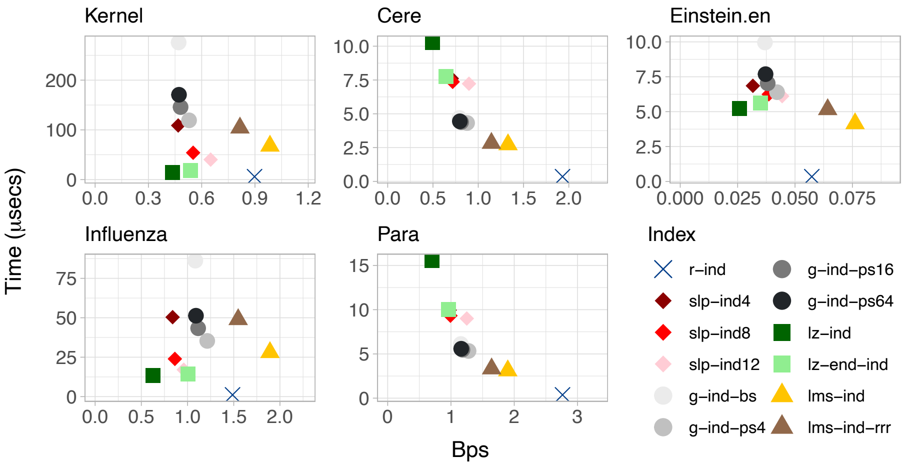
Experiments

- We implemented the grammar algorithm of Christiansen et al. 2020 and compared it with our LMS-based locally consistent algorithm.
- We use the grammar resulted from our algorithm to build the self-index of Claude et al. 2020.
  - We implemented two variations; `lms-ind` and `lms-ind-rrr`
- We compared our version of the grammar self-index against the state-of-the-art dictionary-based self-indexes:
  - `lz-ind`: a self-index based on the Lempel-Ziv parsing technique
  - `slp-ind`: original version of the grammar index that works on straight-line program.
  - `g-ind`: most recent version of the grammar index that works on any type of grammar.
  - `r-ind`: the r-index.
- We assessed the space usage and the time for answering the *locate* operation.

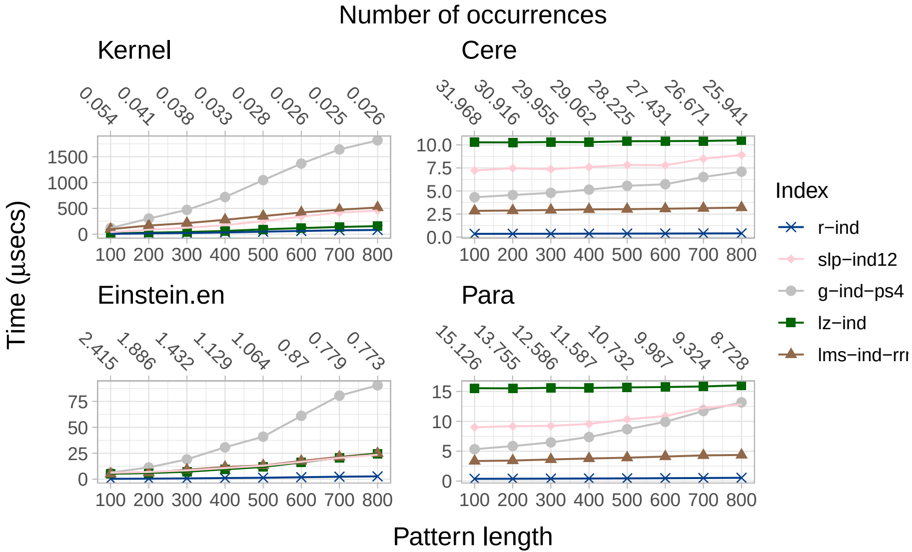
Results: grammar algorithm

Dataset	n	σ	RePair	LMS	LMS post	LC
para	429,265,758	5	5,344,480	22,787,047	8,933,303	8,888,002
cere	461,286,644	5	4,069,450	37,426,507	6,802,801	4,069,450
influenza	154,808,555	15	1,957,370	4,259,746	3,304,035	4,477,322
einstein.en	467,626,544	139	212,903	643,338	427,142	601,755
kernel	257,961,616	162	1,374,650	3,769,839	2,870,350	3,795,801

Results: self-indexes



Results: locate operation



Further work

- Reduce the space usage of the self-index:

Further work

- Reduce the space usage of the self-index:
  - How can we further reduce the grammar size without losing local consistency?

Further work

- Reduce the space usage of the self-index:
  - How can we further reduce the grammar size without losing local consistency?
  - Is it possible to use the Wheeler graph framework to create a grammar-based self-index?

Further work

- Reduce the space usage of the self-index:
  - How can we further reduce the grammar size without losing local consistency?
  - Is it possible to use the Wheeler graph framework to create a grammar-based self-index?
- Can we use the concept of locally consistent parsing to support inexact locate queries?

Further work

- Reduce the space usage of the self-index:
  - How can we further reduce the grammar size without losing local consistency?
  - Is it possible to use the Wheeler graph framework to create a grammar-based self-index?
- Can we use the concept of locally consistent parsing to support inexact locate queries?
- What other types of queries can we support using local consistency?

Questions?