U-index: A Universal Indexing Framework for Matching Long Patterns

Giulio Ermanno Pibiri

Ca' Foscari University of Venice

@jermp.bsky.social

@jermp

23-rd Symposium on Experimental Algorithms (SEA 2025)
Venice, Italy, 22 July 2025

Joint work with
Lorraine A. K. Ayad
Gabriele Fici
Ragnar Groot Koerkamp
Grigorios Loukides
Rob Patro
Solon P. Pissis

The text indexing problem

- Given a string $T[0..n)$ over the alphabet $\Sigma = [0..\sigma)$, pre-process T so that the following queries can be answered efficiently for any string $P[0..m)$:

  - $\mathit{Locate}(P, T)$: return all the positions where P occurs in T;
  - $\mathit{Count}(P, T)$: count the number of occurrences of P in T;
  - $\mathit{Extract}(i, j, T)$: report the substring $T[i\,..\,j]$.

- Fundamental and well-studied problem.

The text indexing problem

- Given a string $T[0..n)$ over the alphabet $\Sigma = [0..\sigma)$, pre-process T so that the following queries can be answered efficiently for any string $P[0..m)$:

  - $Locate(P, T)$: return all the positions where P occurs in T;
  - $Count(P, T)$: count the number of occurrences of P in T;
  - $Extract(i, j, T)$: report the substring $T[i..j]$.

- Fundamental and well-studied problem.

- **More about this on Friday: “25 years of compressed self-indexes” !**

The text indexing problem

- Many solutions with different trade-offs between space and time.
- Solutions broadly fall into two categories:

The text indexing problem

- Many solutions with different trade-offs between space and time.
- Solutions broadly fall into two categories:

    1. **Compressed**: The text is replaced (is “self-indexed”) with a compressed representation.
    2. **Uncompressed**: A redundancy (an “index”) is attached to T to accelerate queries.

The text indexing problem

- Many solutions with different trade-offs between space and time.
- Solutions broadly fall into two categories:

    1. **Compressed**: The text is replaced (is “self-indexed”) with a compressed representation.

    2. **Uncompressed**: A redundancy (an “index”) is attached to T to accelerate queries.

- Solutions in 1. are very space-efficient but generally slower to build and query than solutions in 2. which — on the other hand — are space-inefficient.
- **Example.** The (uncompressed) **suffix array** is much faster to query than the FM-index but requires $O(n \log n)$ bits on top of the text.

Our contribution

- We focus on the uncompressed case, i.e., we attach an index to the text, and address the space-inefficiency of the index while supporting efficient queries.

Our contribution

- We focus on the uncompressed case, i.e., we attach an index to the text, and address the space-inefficiency of the index while supporting efficient queries.

- **Main idea**: if we compute a **sketch** of the text T, say $S = Sketch(T)$, then $Index(S)$ will be smaller/faster than $Index(T)$ because S is **a lot smaller** than T, for any $Index$.

- At query time: we also compute $Q = Sketch(P)$ and match Q against S. Candidate matches (including *false positives*) are mapped back to T to be verified.

Our contribution

- We focus on the uncompressed case, i.e., we attach an index to the text, and address the space-inefficiency of the index while supporting efficient queries.

- **Main idea**: if we compute a **sketch** of the text T, say $S = Sketch(T)$, then $Index(S)$ will be smaller/faster than $Index(T)$ because S is **a lot smaller** than T, for any $Index$.

- At query time: we also compute $Q = Sketch(P)$ and match Q against S. Candidate matches (including *false positives*) are mapped back to T to be verified.

- That is, we have a **universal framework** because:

  - **any index** can be used for S;

  - **any locally-consistent sampling algorithm** can be used to sketch the text and obtain S.

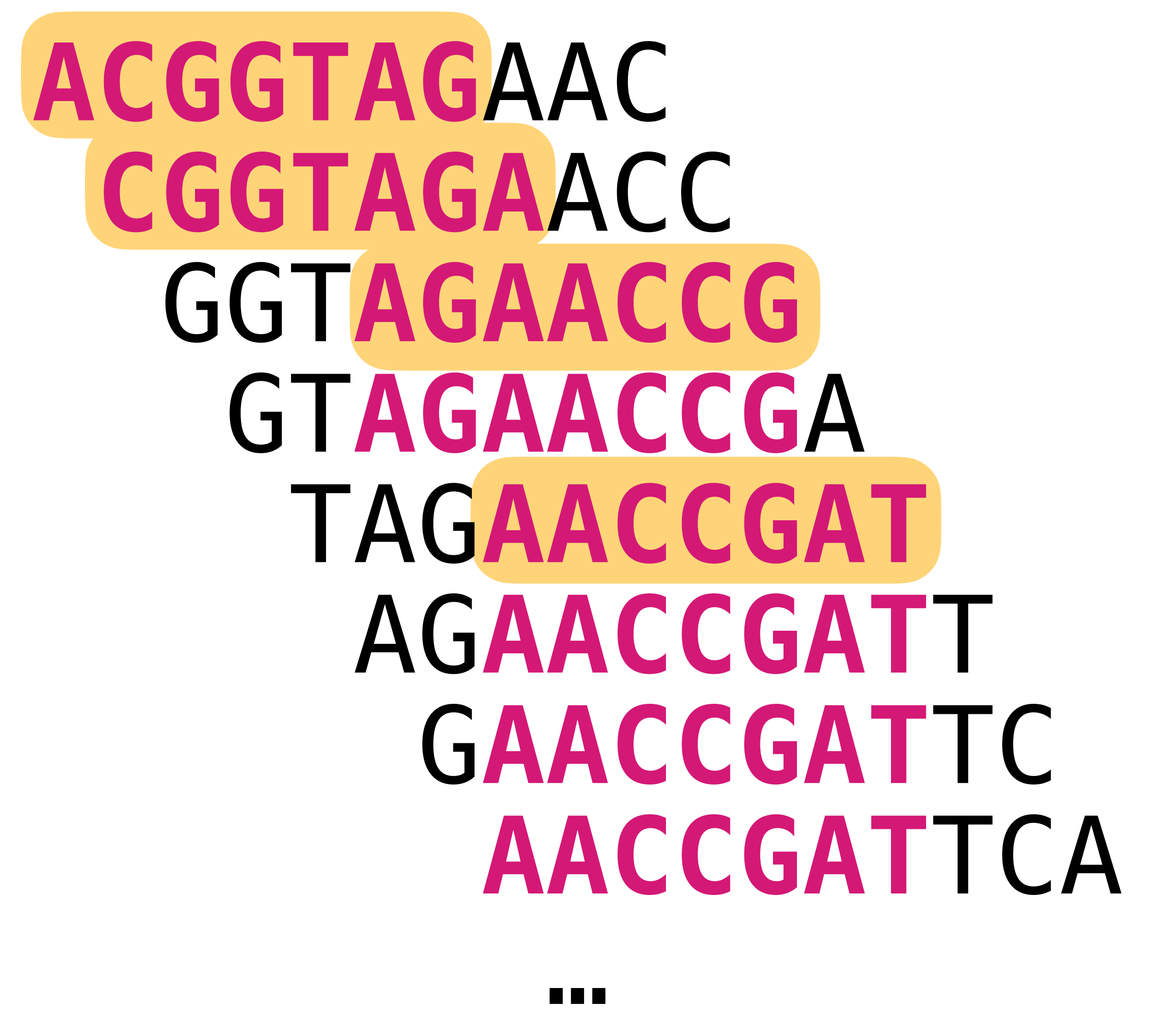
Intermezzo: sketching with minimizers

- Consider each window of w consecutive k-mers from a string T: sample one k-mer out of w and call it the “representative” of the window — or its *minimizer*.

- We would like to sample the **same minimizer** from consecutive windows so that the **set of distinct minimizers** forms a succinct sketch for T.

Example for $w = 4$ and $k = 7$.

```
ACGGTAGAACCGATTCAAATTCGAT…
```



```
ACGGTAGAAC
 CGGTAGAACC
  GGTAGAACCG
   GTAGAACCGA
    TAGAACCGAT
     AGAACCGATT
      GAACCGATTC
       AACCGATTCA
          …
```

Intermezzo: sketching with minimizers

- **Q.** How do we compare different sampling algorithms?

  A. We define the *density* of a sampling algorithm as the fraction between the number of (distinct) minimizers and the total number of k-mers of T.

  The lower the density, the better!

Intermezzo: sketching with minimizers

- **Q.** How do we compare different sampling algorithms?

  A. We define the *density* of a sampling algorithm as the fraction between the number of (distinct) minimizers and the total number of k-mers of T.

  The lower the density, the better!

- Since the same k-mer cannot be a minimizer for more than w consecutive k-mers, we immediately have a **lower bound** of $1/w$ on the density of any sampling algorithm.

```
TAGAACCGAT
 AGAACCGATT
  GAACCGATTC
   AACCGATTCA
    ...
```

The “folklore”, random, minimizer

```
1: function MINIMIZER($W, w, k, \mathcal{O}_k$)
2:     $o_{min} = +\infty$
3:     $p = 0$
4:     for $i = 0$; $i < w$; $i = i + 1$ do
5:         $o = \mathcal{O}_k(W[i..i + k))$
6:         if $o < o_{min}$ then
7:             $o_{min} = o$
8:             $p = i$
9:     return $p$
```

Example for $w = 4$ and $k = 7$.

```
ACGGTAGAACCGATTCAAATTCGAT…

ACGGTAGAAC
 CGGTAGAACC
  GGTAGAACCG
   GTAGAACCGA
    TAGAACCGAT
     AGAACCGATT
      GAACCGATTC
       AACCGATTCA
         …
```

- We usually define the total order using a random hash function (*random* minimizer).
- In this case, the density is $2/(w + 1)$: almost a factor of 2 away from the lower bound for large w.

The “folklore”, random, minimizer

```
1: function MINIMIZER($W, w, k, \mathcal{O}_k$)
2:     $o_{min} = +\infty$
3:     $p = 0$
4:     for $i = 0; i < w; i = i + 1$ do
5:         $o = \mathcal{O}_k(W[i..i + k))$
6:         if $o < o_{min}$ then
7:             $o_{min} = o$
8:             $p = i$
9:     return $p$
```

Example for $w = 4$ and $k = 7$.

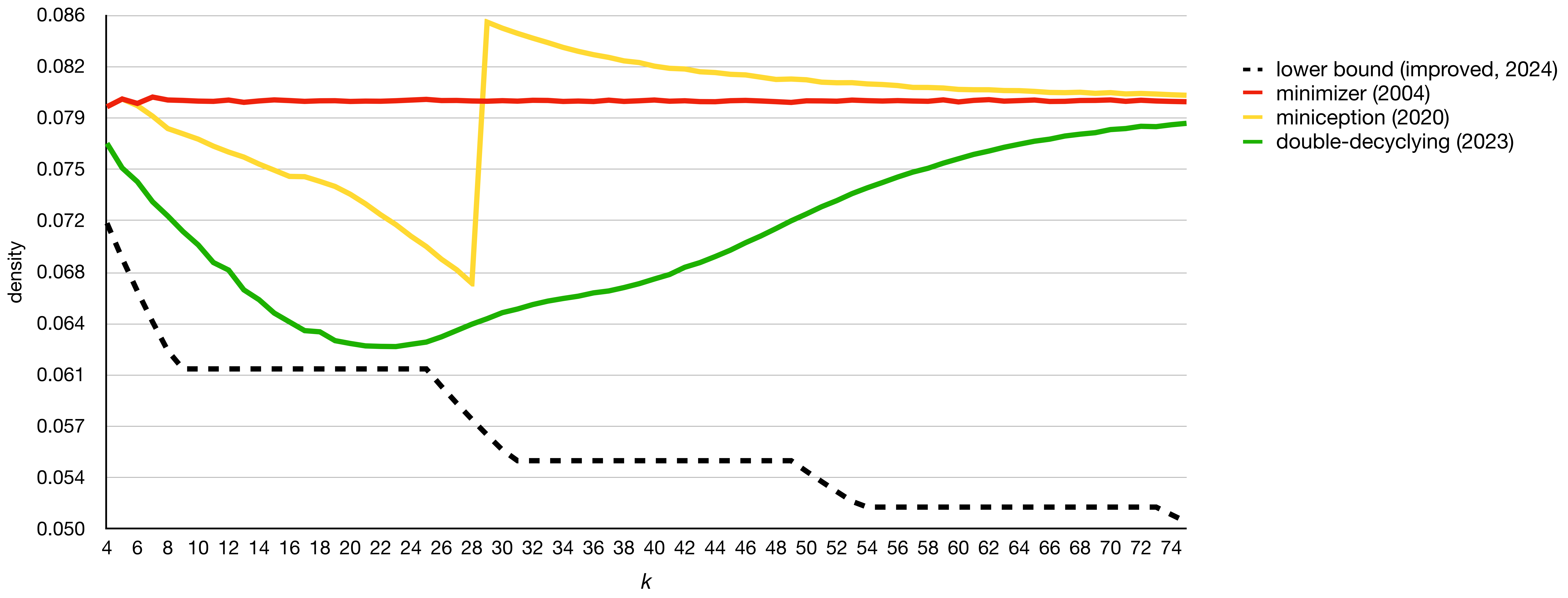
```
ACGGTAGAACCGATTCAAATTCGAT…

ACGGTAGAAC
 CGGTAGAACC
  GGTAGAACCG
   GTAGAACCGA
    TAGAACCGAT
     AGAACCGATT
      GAACCGATTC
       AACCGATTCA
        …
```

- We usually define the total order using a random hash function (*random* minimizer).
- In this case, the density is $2/(w+1)$: almost a factor of 2 away from the lower bound for large w.

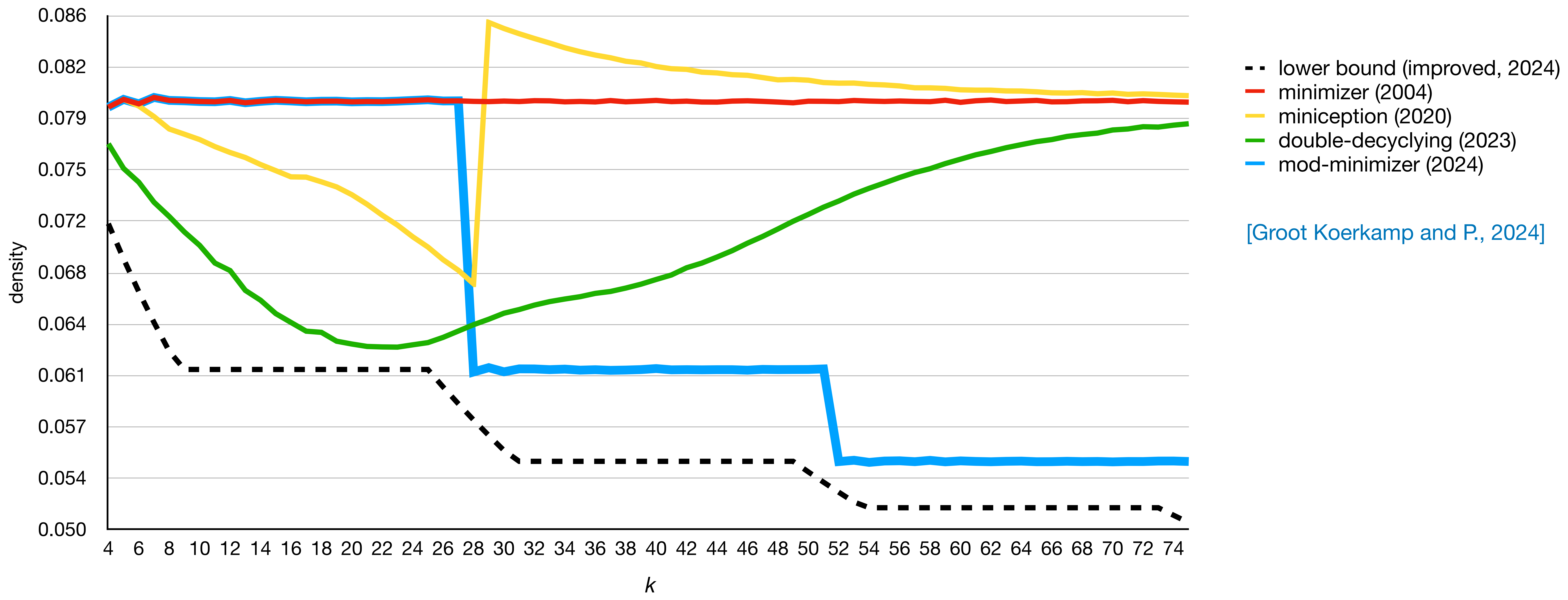
More about them on Thursday!

Density by varying *k*



- Example for $w = 24$.
- Measured over a string of 10 million i.i.d. random characters with an alphabet size of 4.
- https://github.com/jermp/minimizers

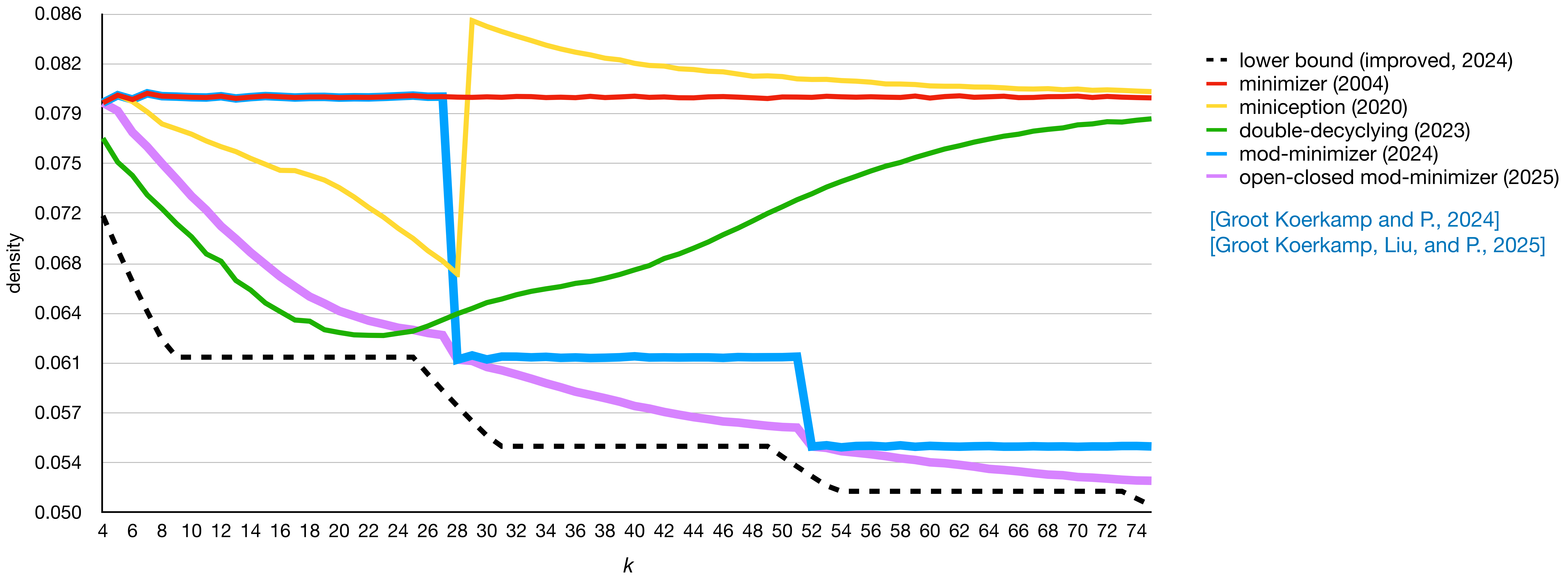
Density by varying *k*



[Groot Koerkamp and P., 2024]

- Example for $w = 24$.
- Measured over a string of 10 million i.i.d. random characters with an alphabet size of 4.
- https://github.com/jermp/minimizers

Density by varying *k*

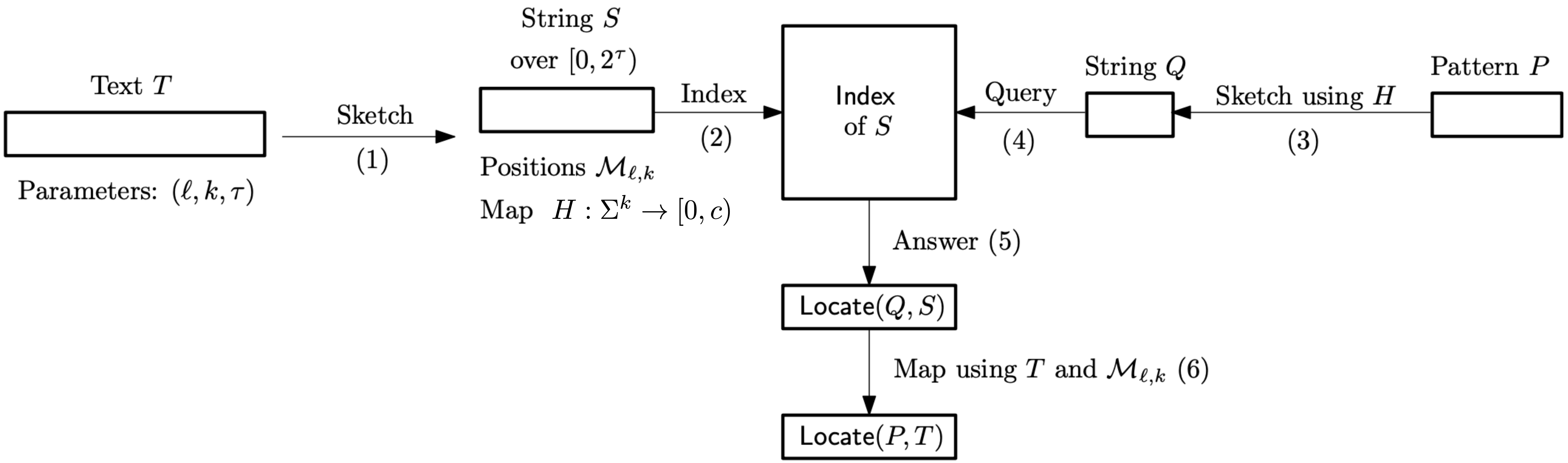


[Groot Koerkamp and P., 2024]
[Groot Koerkamp, Liu, and P., 2025]

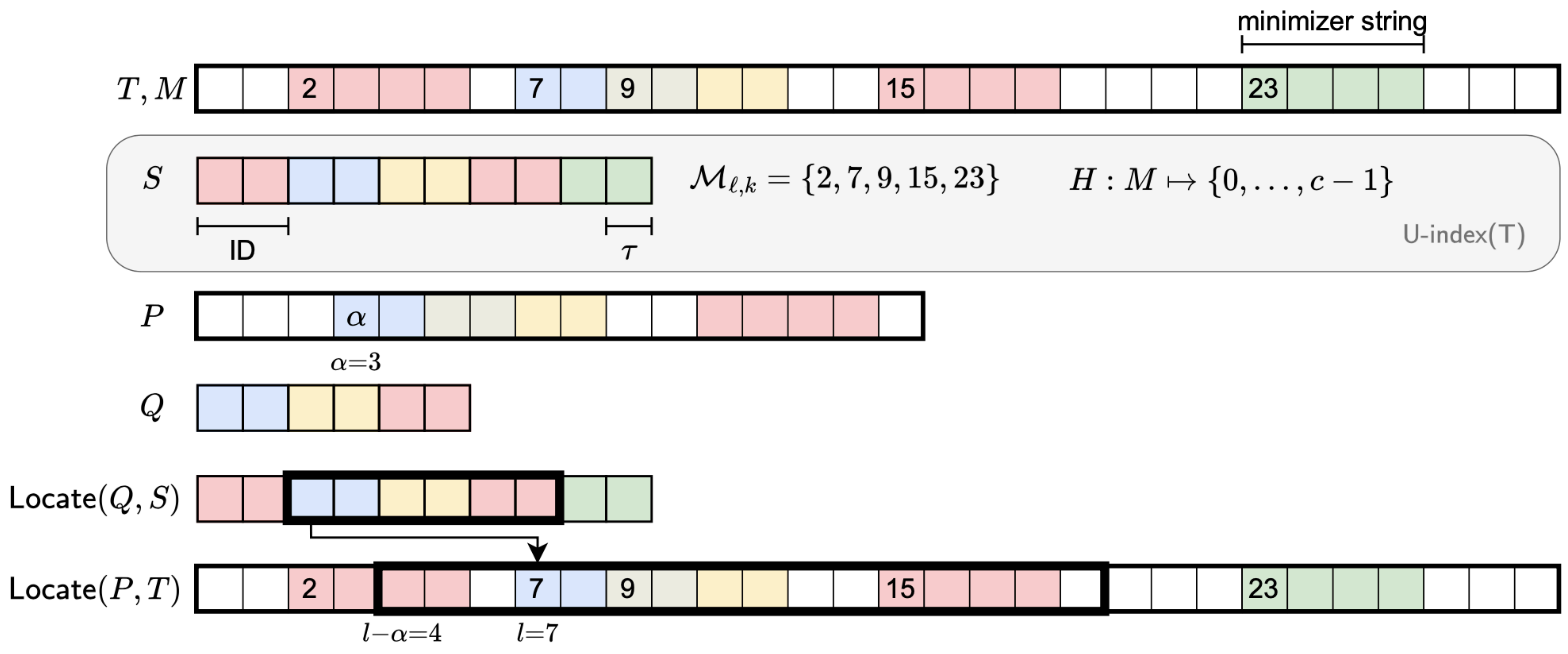
- Example for $w = 24$.
- Measured over a string of 10 million i.i.d. random characters with an alphabet size of 4.
- https://github.com/jermp/minimizers

The U-index framework for matching long patterns

- We fix integers $\ell > 0$ and $k > 0$ and let $w := \ell - k + 1$, so that any pattern P of length $m \geq \ell$ contains at least one minimizer.

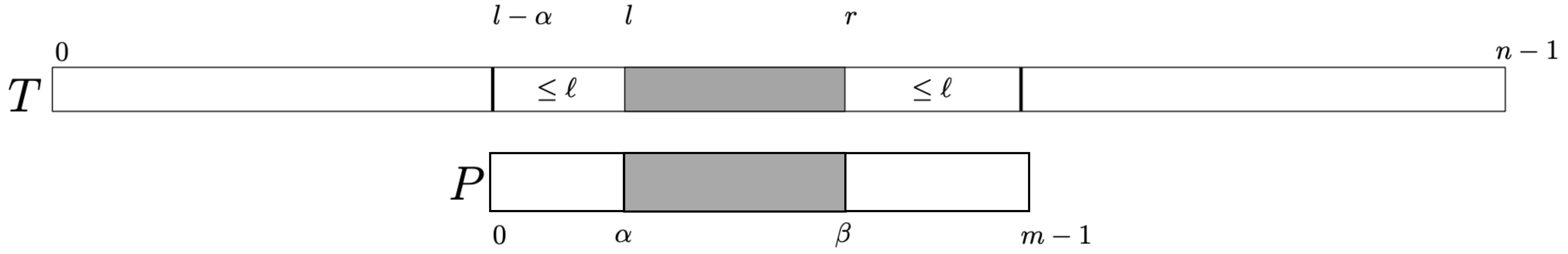

Text T
Parameters: (ℓ, k, τ)
Sketch
(1)
String S over $[0, 2^\tau)$
Positions $\mathcal{M}_{\ell,k}$
Map $H : \Sigma^k \to [0, c)$
Index
(2)
Index of S
Query
(4)
String Q
Sketch using H
(3)
Pattern P
Answer (5)
Locate(Q, S)
Map using T and $\mathcal{M}_{\ell,k}$ (6)
Locate(P, T)

An example


minimizer string
T, M
2
7
9
15
23
S
ID
τ
$\mathcal{M}_{\ell,k} = \{2, 7, 9, 15, 23\}$
$H : M \mapsto \{0, \ldots, c-1\}$
U-index(T)
P
α
α=3
Q
Locate(Q, S)
Locate(P, T)
$l-\alpha$=4
l=7

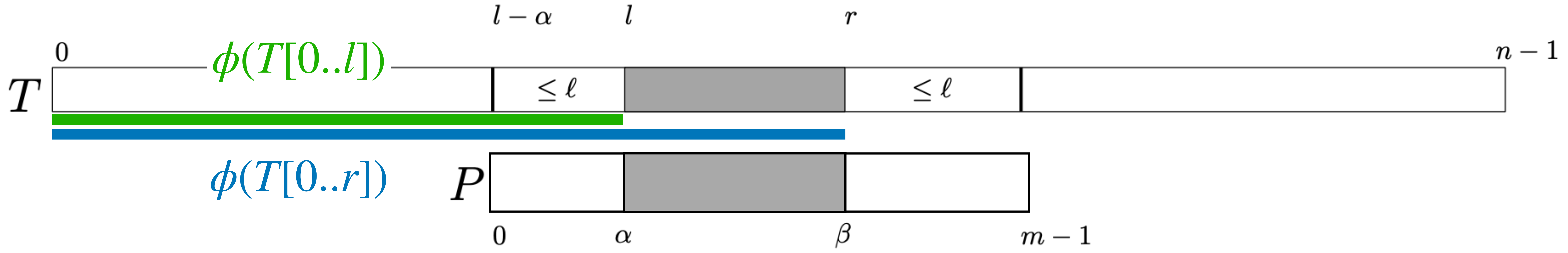
Theoretical guarantees

- Using some machinery, we guarantee that an occurrence in $Locate(Q, S)$ is verified in $O(1)$, rather than $O(m)$. This can be done in $O(z)$ space on top of the space of the text, where z is the number of minimizers of T.



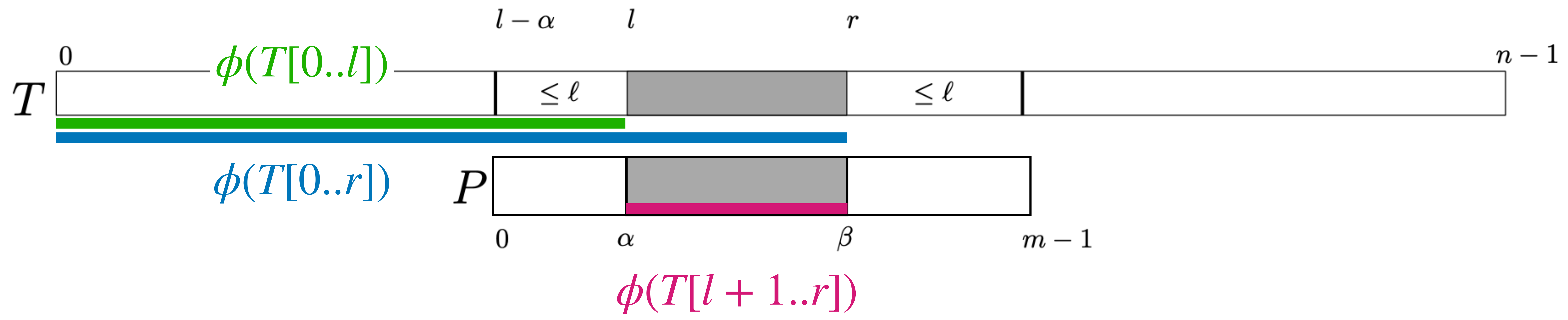
Theoretical guarantees

- Using some machinery, we guarantee that an occurrence in $Locate(Q, S)$ is verified in $O(1)$, rather than $O(m)$. This can be done in $O(z)$ space on top of the space of the text, where z is the number of minimizers of T.



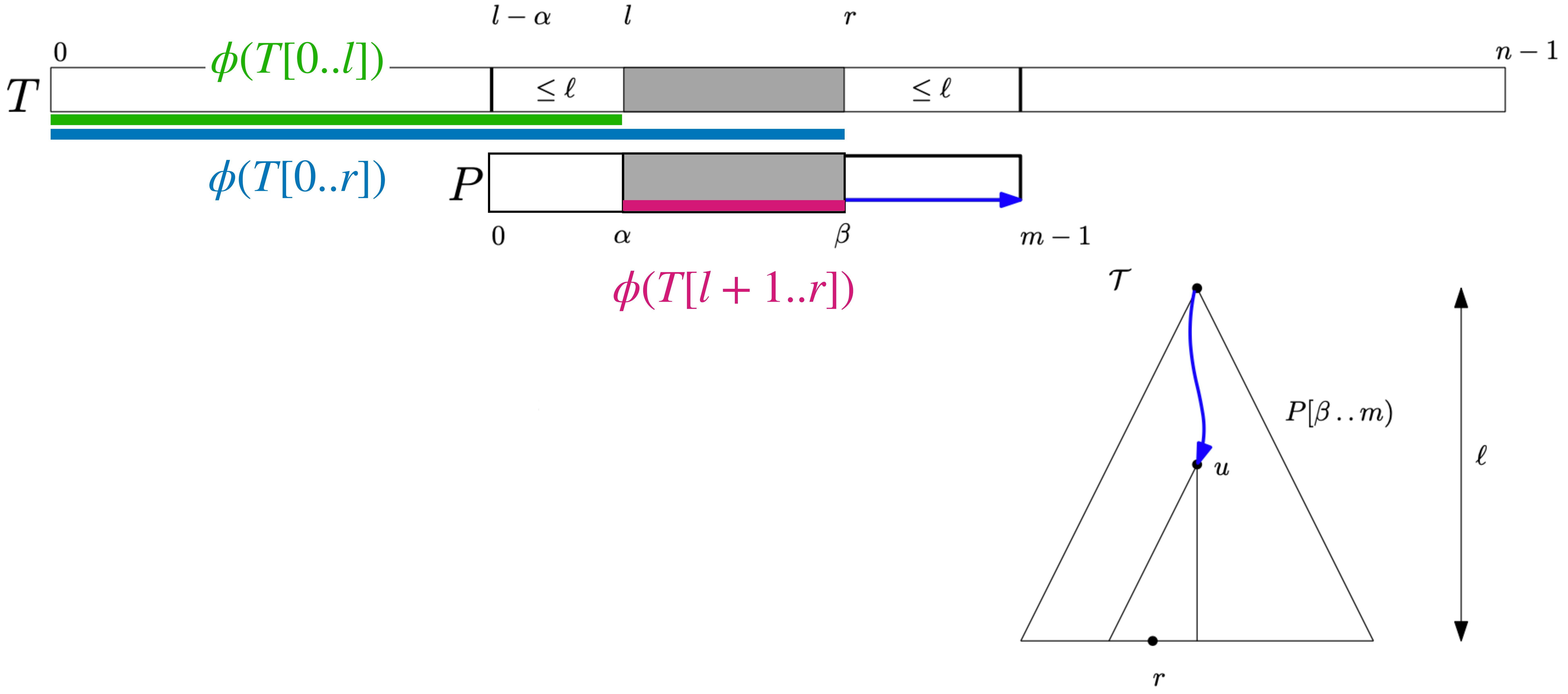
Theoretical guarantees

- Using some machinery, we guarantee that an occurrence in $Locate(Q, S)$ is verified in $O(1)$, rather than $O(m)$. This can be done in $O(z)$ space on top of the space of the text, where z is the number of minimizers of T.



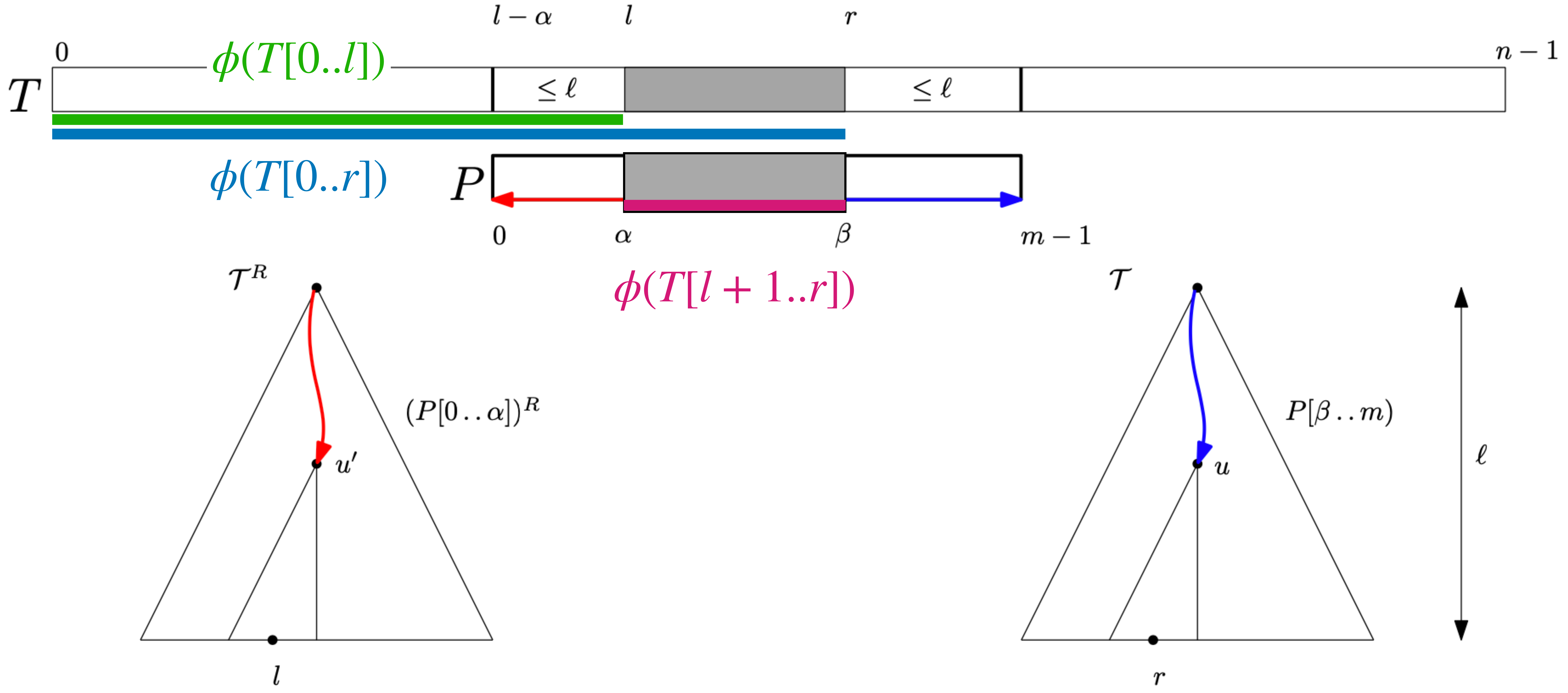
Theoretical guarantees

- Using some machinery, we guarantee that an occurrence in $Locate(Q, S)$ is verified in $O(1)$, rather than $O(m)$. This can be done in $O(z)$ space on top of the space of the text, where z is the number of minimizers of T.

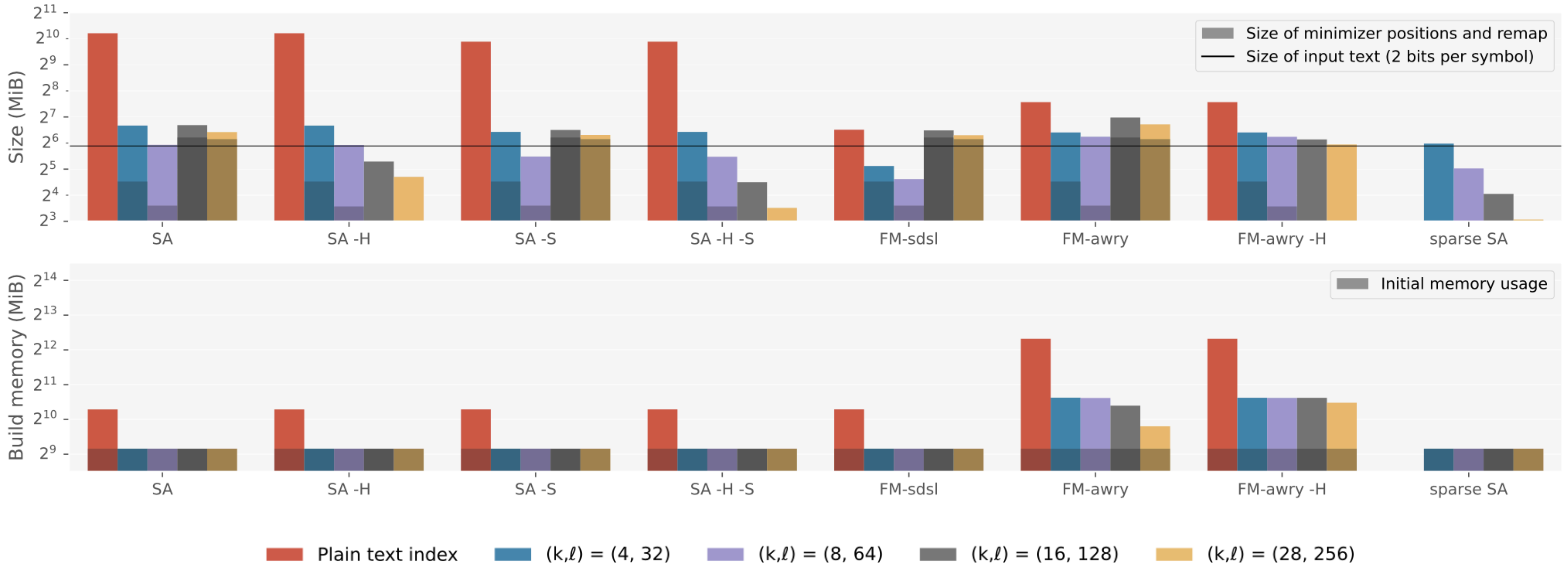


Theoretical guarantees

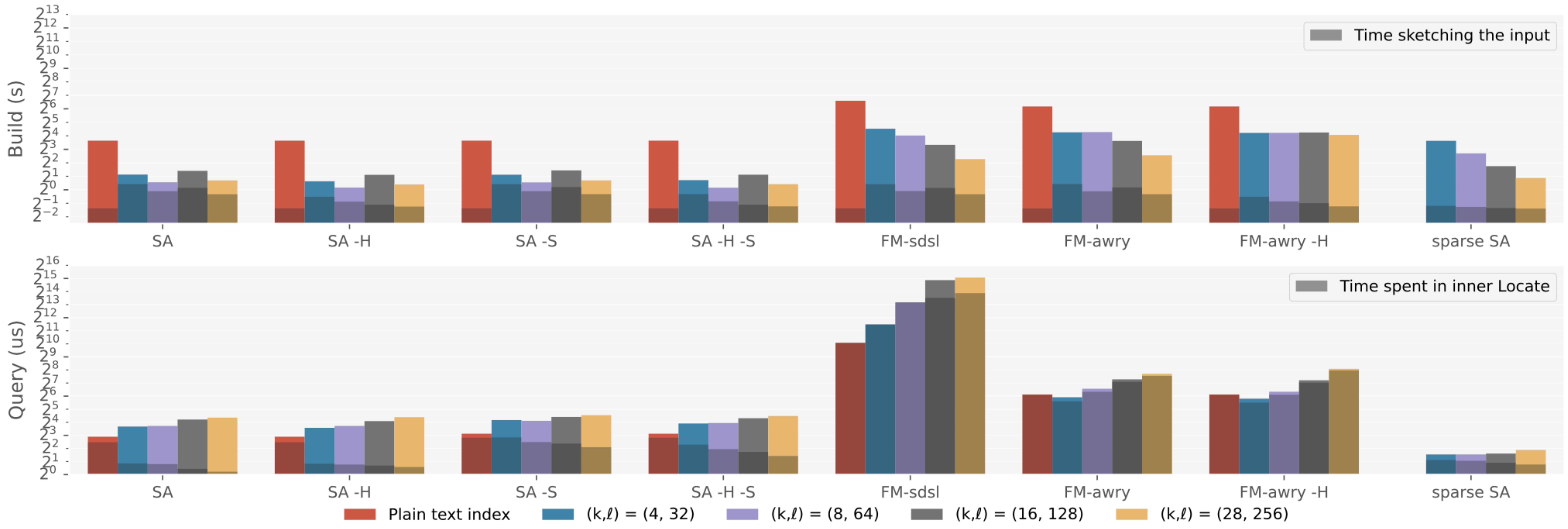
- Using some machinery, we guarantee that an occurrence in $Locate(Q, S)$ is verified in $O(1)$, rather than $O(m)$. This can be done in $O(z)$ space on top of the space of the text, where z is the number of minimizers of T.



Results — Index size and build space for human chr 1



Results — Build and query time for human chr 1

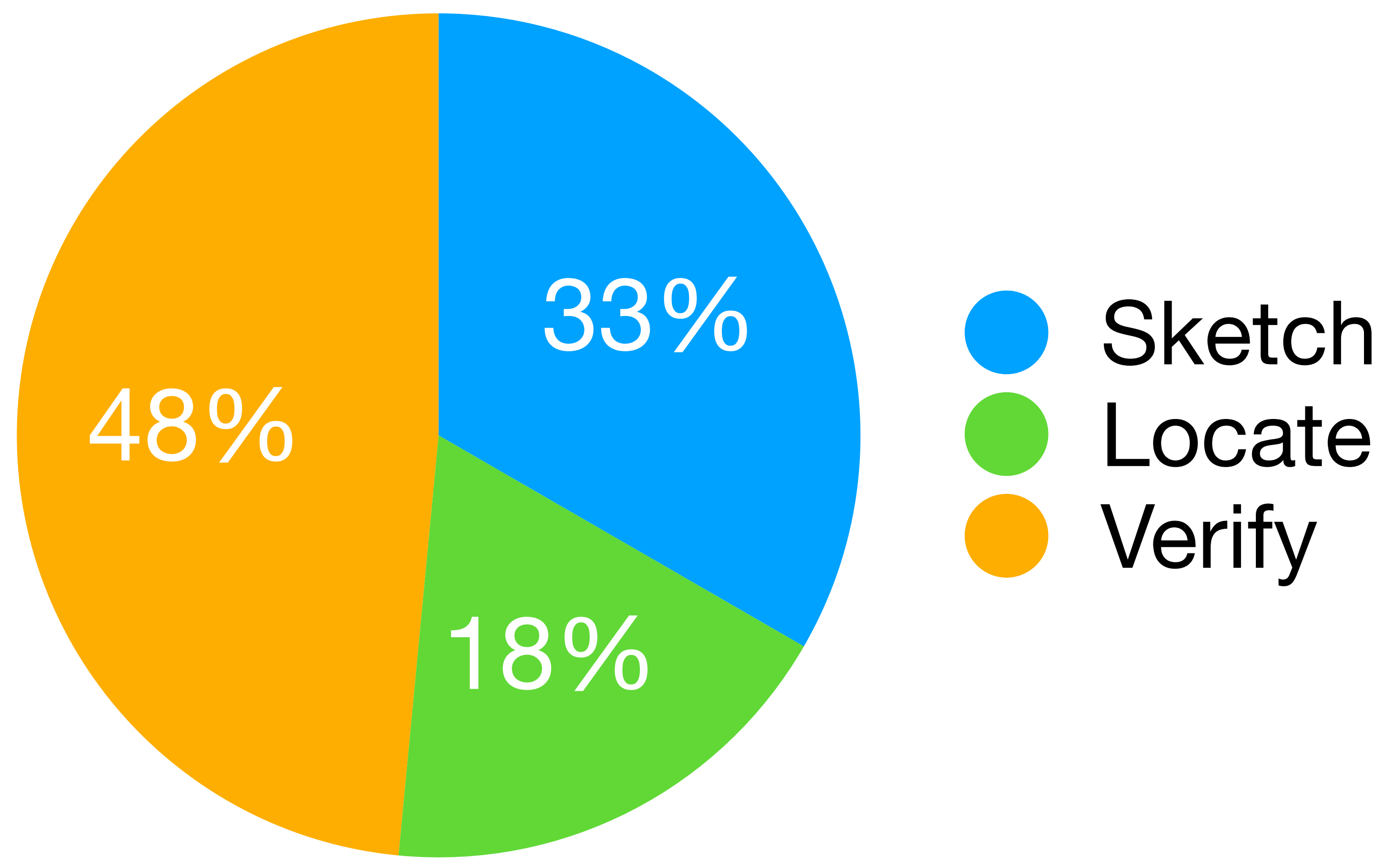


An example application

- We demonstrated that the U-index framework can be useful for **long read mapping**.
- A core problem in Computational Biology; it involves aligning long patterns to a reference genome.
- Experimental setting: we align 450 HiFi long reads (avg. length is 16 kbp) on a complete human reference genome. We use $k = 8$ and $\ell = 128$.

An example application

- We demonstrated that the U-index framework can be useful for **long read mapping**.
- A core problem in Computational Biology; it involves aligning long patterns to a reference genome.
- Experimental setting: we align 450 HiFi long reads (avg. length is 16 kbp) on a complete human reference genome. We use $k = 8$ and $\ell = 128$.
- Very practical numbers **using a suffix array** as index: the U-index is built in 12 seconds with $\approx 9\mu$s per pattern (23 avg. false positives per pattern).



Conclusions

- Main take-away: U-index is a framework to **enhance the performance of any off-the-shelf text index**, provided that the patterns to match are **reasonably long**.
- The framework is very flexible: many space vs. time trade-offs possible depending on the index and sampling scheme used.
- Bottleneck: verifying false positive matches.
- Rust code: https://github.com/u-index/u-index-rs

Conclusions

- Main take-away: U-index is a framework to **enhance the performance of any off-the-shelf text index**, provided that the patterns to match are **reasonably long**.
- The framework is very flexible: many space vs. time trade-offs possible depending on the index and sampling scheme used.
- Bottleneck: verifying false positive matches.
- Rust code: https://github.com/u-index/u-index-rs

Thank you for the attention!
A special thank to all my co-authors!