Locality-preserving minimal perfect hashing of k-mers

Giulio Ermanno Pibiri

Ca' Foscari University of Venice, Venice, Italy

@giulio_pibiri

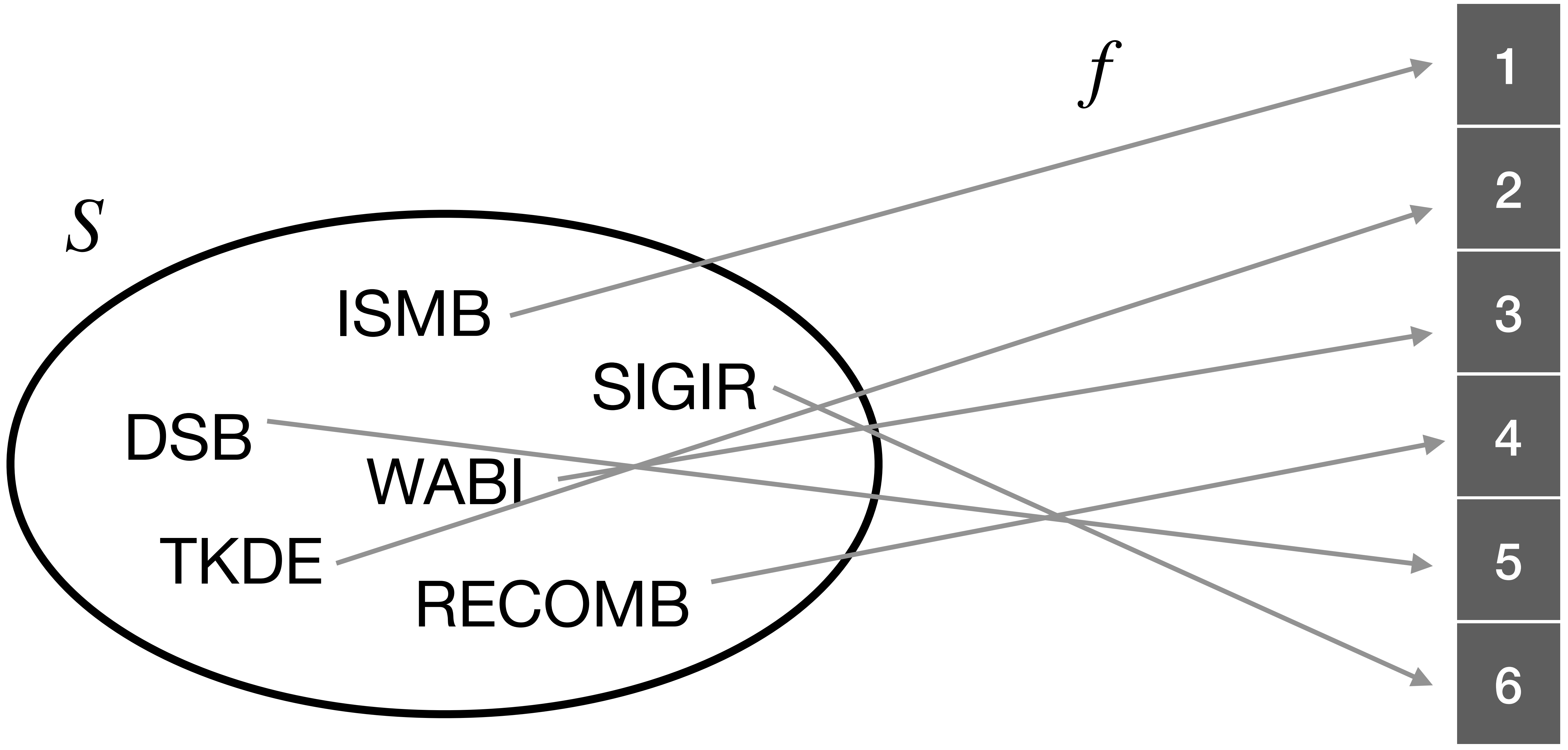
@jermp

Joint work with
Yoshihiro Shibuya
(Institut Pasteur, Paris, France)
and
Antoine Limasset
(CNRS, Lille, France)

ISMB 2023
Lyon, France, 25 July 2023

Minimal perfect hashing

MPHF. Given a set $S \subseteq U$ of n distinct keys, a function $f : U \to \{1,\ldots,n\}$ such that $f(x) \neq f(y)$ for any $x, y \in S$, $x \neq y$, is called a *minimal perfect hash function* (MPHF) for S.



Minimal perfect hashing

- **Q.** How much space do we need to represent a MPHF?

- **A.** Lower bound of $\log_2(e) \approx 1.442$ bits/key [Mehlhorn, 1982].
- In practice: 2 – 4 bits/key and **constant** evaluation time.
- Many algorithms are known for minimal perfect hashing.

  - FCH [Fox et al., 1992]
  - CHD [Belazzougui et al., 2009]
  - EMPHF [Belazzougui et al., 2014]
  - GOV [Genuzio et al., 2016]
  - BBHash [Limasset et al., 2017]
  - RecSplit [Esposito et al., 2019]
  - PTHash [P. and Trani, 2021]
  - SicHash [Lehmann et al., 2023]
  - FMPHGO [Beling, 2023]

What about specific inputs?

- Note that the $\log_2(e)$ bits/key lower bound is valid for a **generic input** set S and, as such, does not exploit any property the keys might have.
- **This does not rule out more succinct solutions if we consider specific inputs.**
- In practice, the keys we hash often present some *intrinsic relationships* that we could exploit to lower the bit-complexity and evaluation time of f.

What about specific inputs?

- Note that the $\log_2(e)$ bits/key lower bound is valid for a **generic input** set S and, as such, does not exploit any property the keys might have.

- **This does not rule out more succinct solutions if we consider specific inputs.**

- In practice, the keys we hash often present some *intrinsic relationships* that we could exploit to lower the bit-complexity and evaluation time of f.

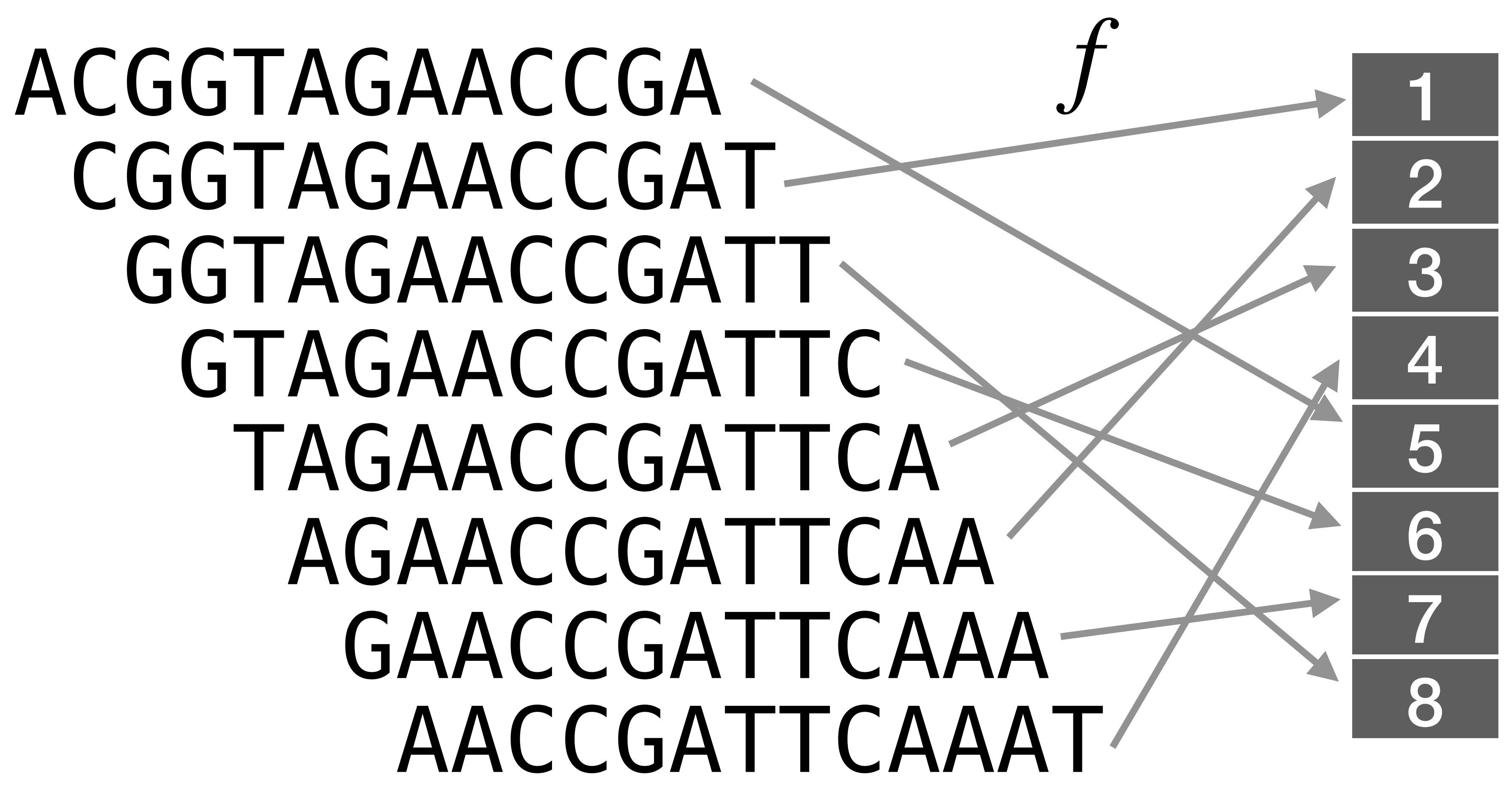
- **Q.** Any example?

What about specific inputs?

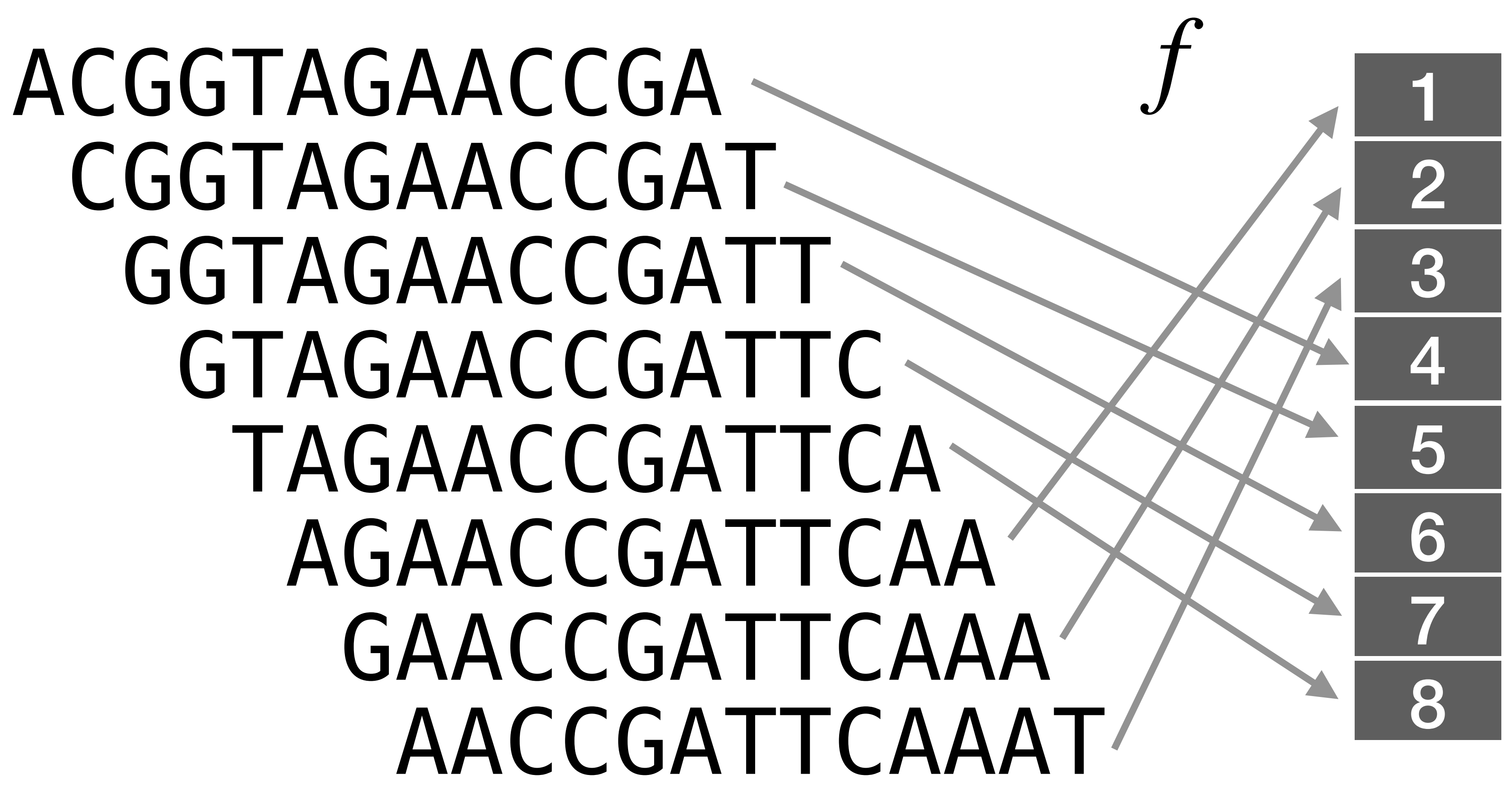
- Note that the $\log_2(e)$ bits/key lower bound is valid for a **generic input** set S and, as such, does not exploit any property the keys might have.
- **This does not rule out more succinct solutions if we consider specific inputs.**
- In practice, the keys we hash often present some *intrinsic relationships* that we could exploit to lower the bit-complexity and evaluation time of f.
- **Q.** Any example?
- **A. k-mer sets!** Keys are strings of fixed length k, sharing $(k-1)$-base overlaps.

Hashing k-mer sets

- **Goal.** Given a set S of n distinct k-mers, **preserve** as much as possible the **local relationships between consecutive k-mers** in the codomain.



Generic MPHF f



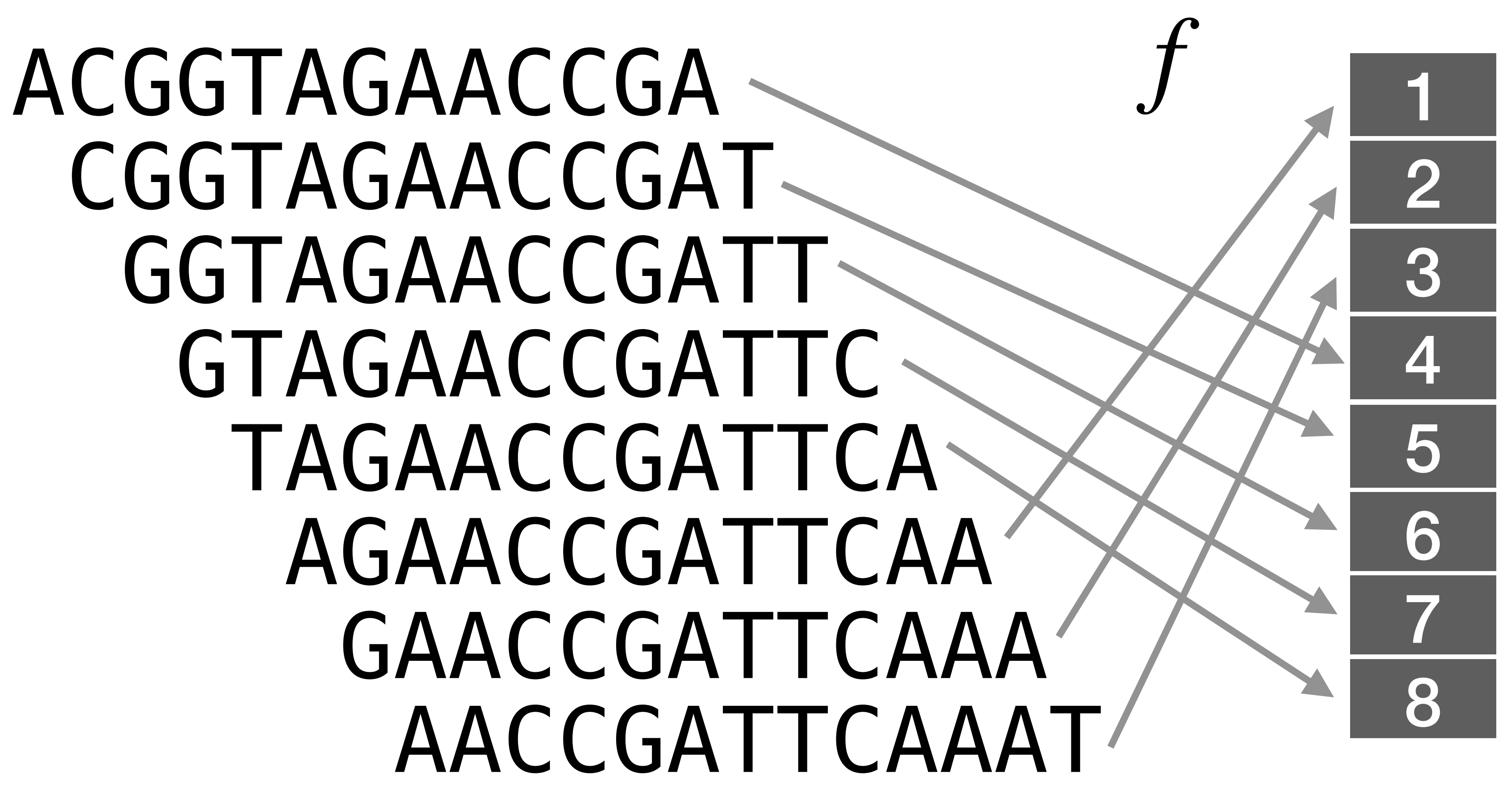
Locality-preserving MPHF f

Hashing k-mer sets

- This behaviour is very desirable as it implies:
  - **Compression of k-mer satellite data.**

    (Abundance counts, dBG unitig identifiers, color classes, etc.) Consecutive k-mers tend to have similar — *if not identical* — satellite data. Locality-preservation induces a natural clustering effect on the satellite values, which aids compression.
  - **Faster access time.**

    Enhanced locality of access when streaming over consecutive k-mers: the next slot to access will be already in cache.



Locality-preserving MPHF f

An example application

- **Problem.** *(Experiment-discovery)* Given a collection of references $\mathscr{R} = \{R_1, \ldots, R_N\}$, how to retrieve the set of references where a given k-mer appears, with false positives allowed?

An example application

- **Problem.** *(Experiment-discovery)* Given a collection of references $\mathscr{R} = \{R_1, \ldots, R_N\}$, how to retrieve the set of references where a given k-mer appears, with false positives allowed?

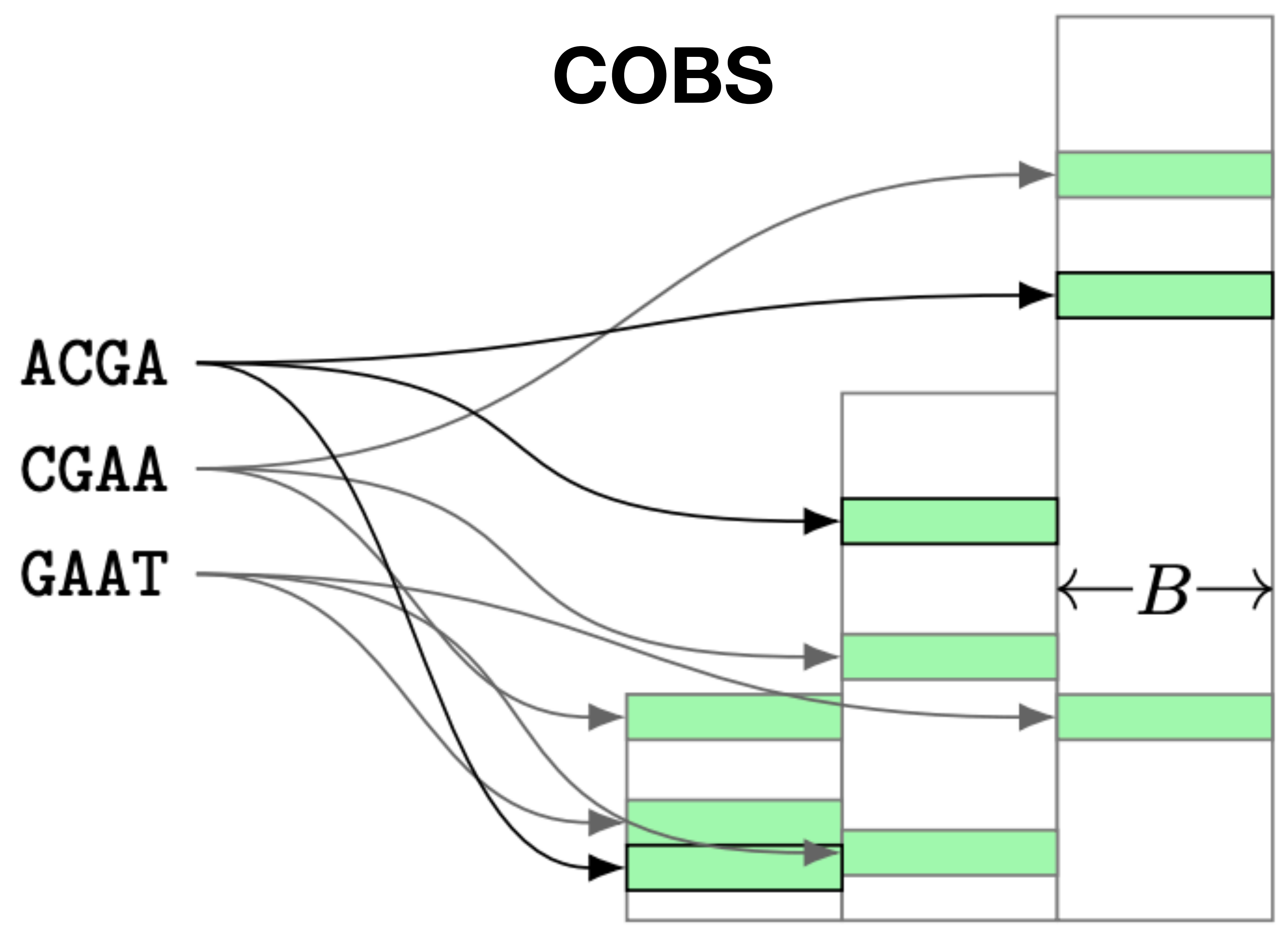


Fig. 5 from [Bingmann et. al, 2019]
https://arxiv.org/pdf/1905.09624.pdf

An example application

- **Problem.** *(Experiment-discovery)* Given a collection of references $\mathscr{R} = \{R_1, \ldots, R_N\}$, how to retrieve the set of references where a given k-mer appears, with false positives allowed?

COBS

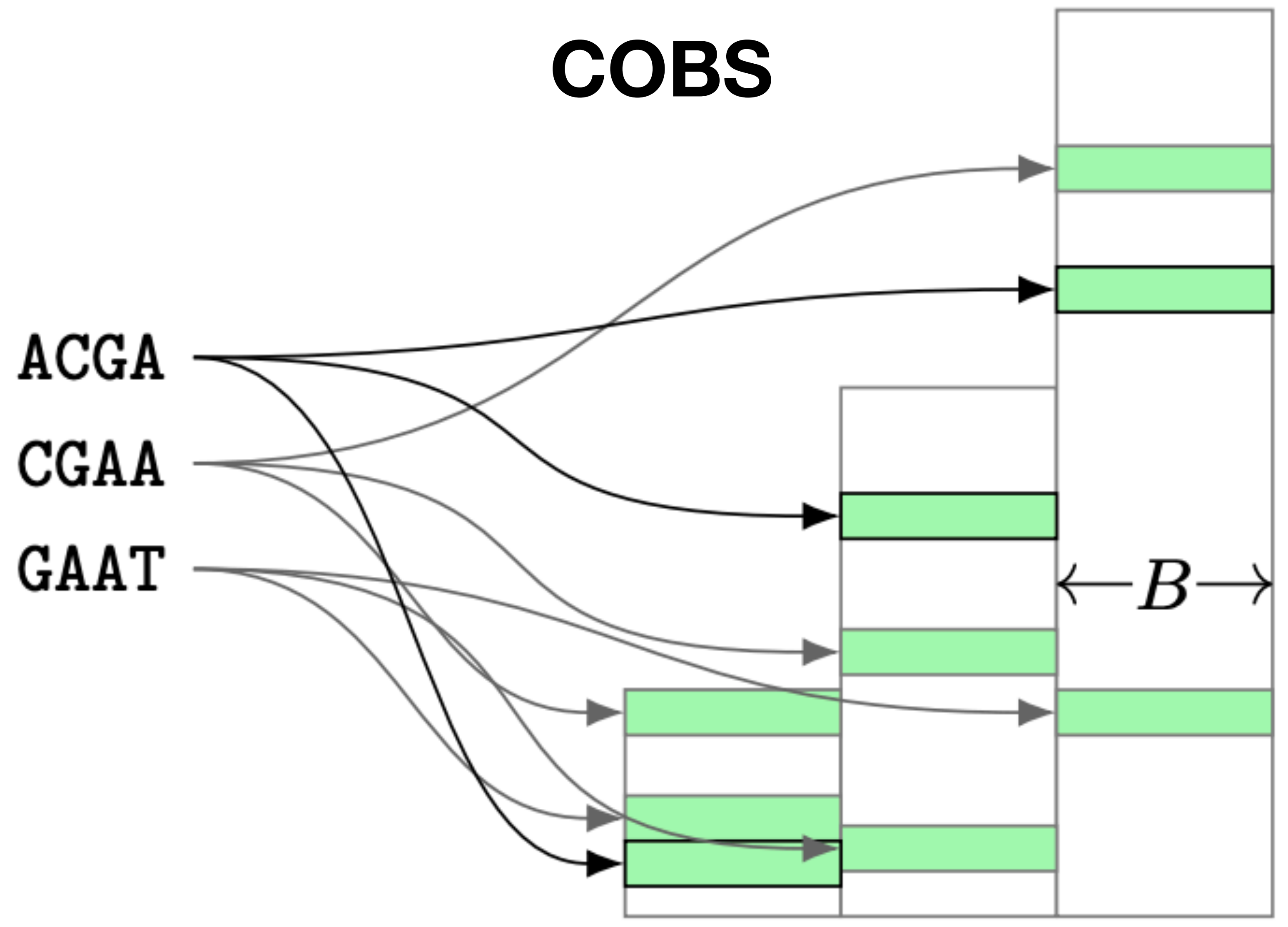
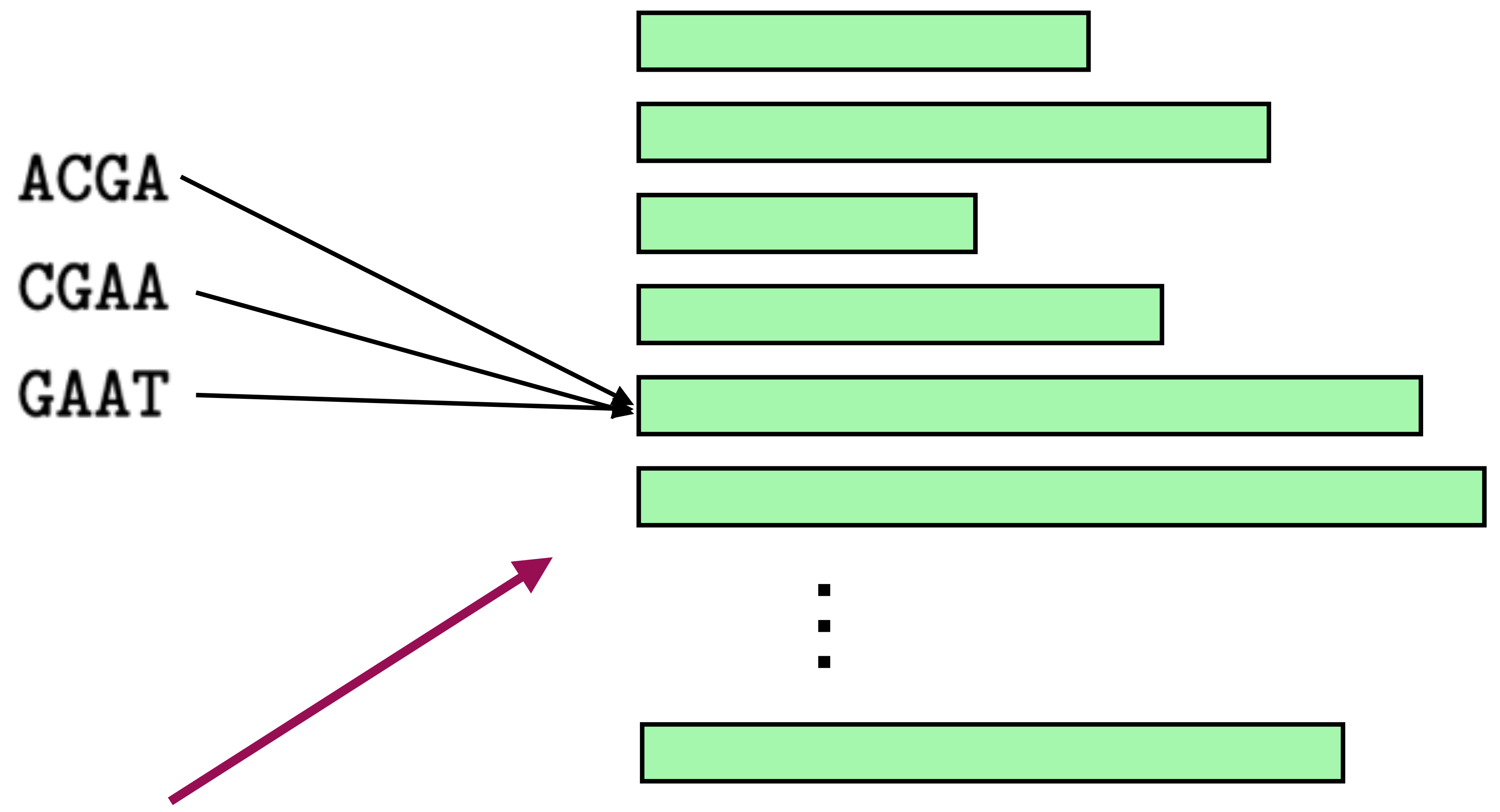


Fig. 5 from [Bingmann et. al, 2019]
https://arxiv.org/pdf/1905.09624.pdf

with locality-preserving hashing

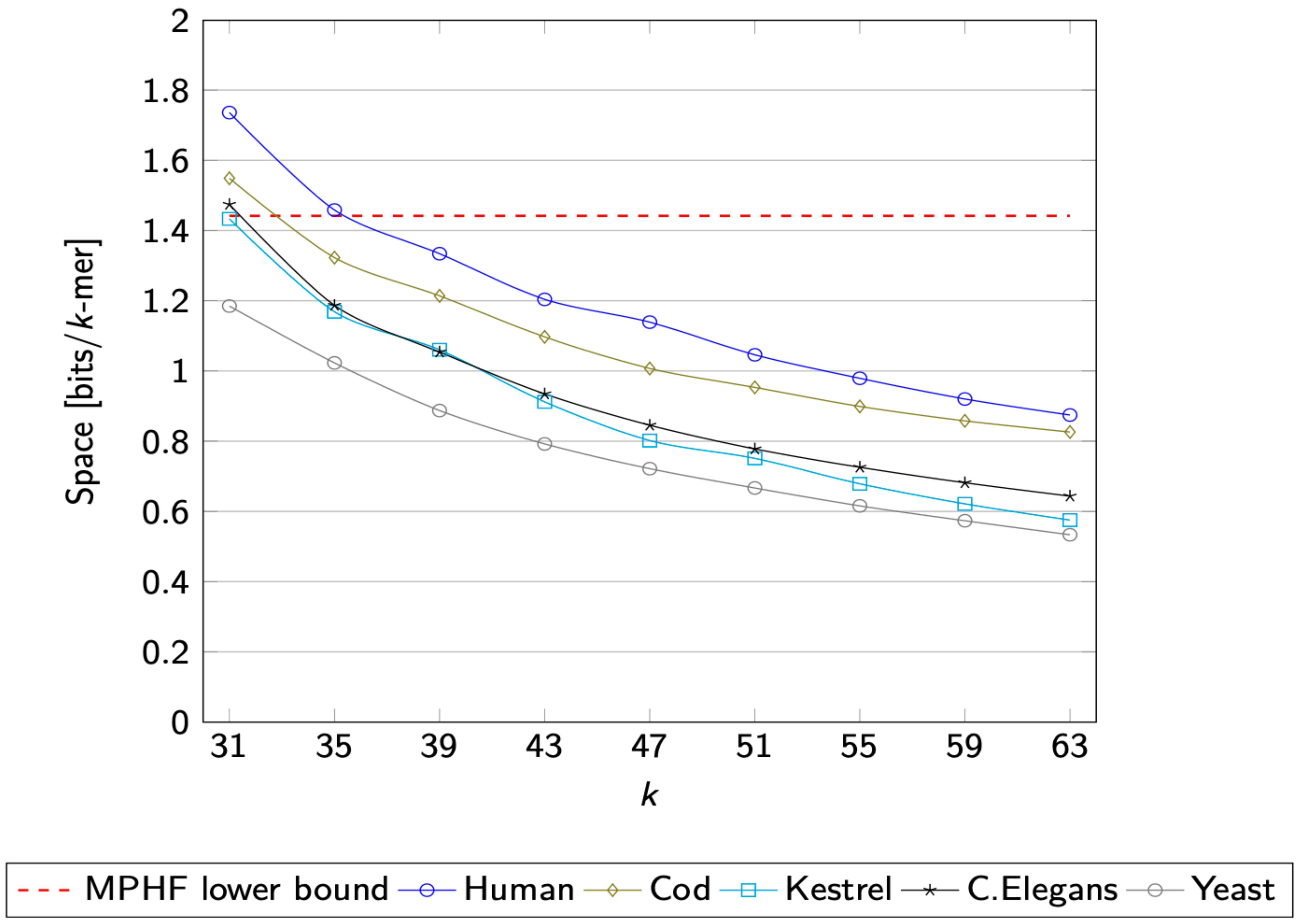


distinct, compressed,
inverted lists (no false positives)

Locality-Preserving (LP) Hash — Overview

- **Smaller** than the classic $\log_2(e) \approx 1.442$ bits/key lower bound on k-mer sets.
  (Check out the paper for the new theoretical characterisation.)
- **Space decreases** when increasing k.
- **Faster streaming query** time compared to the fastest MPHFs (i.e., PTHash).
  Streaming query time decreases when increasing k.
- Scale to billions of k-mers.
- LPHash code in C++ available at https://github.com/jermp/lphash.
- Datasets used in the paper on Zenodo at https://doi.org/10.5281/zenodo.7239205.

LPHash — Space



BBHash size ($\gamma = 1$) = 3.06 bits/k-mer, PTHash size ($\alpha = 0.99$, $c = 5.0$) $\approx$ 2.1 bits/k-mer

LPHash — Query time

Query time in average nanoseconds per k-mer.

Method	k	Yeast		Elegans		Cod		Kestrel		Human	
		stream	random	stream	random	stream	random	stream	random	stream	random
LPHash	31	29	110	40	118	79	144	84	145	107	162
	35	28	125	35	124	65	147	69	149	90	166
	39	27	130	32	131	60	149	63	153	82	166
	43	25	137	30	135	52	152	54	155	73	169
	47	24	145	28	143	47	155	49	159	69	172
	51	24	152	28	150	45	159	46	162	63	174
	55	23	157	26	157	41	165	42	167	59	176
	59	23	165	25	165	39	171	39	173	57	182
	63	22	174	24	172	37	180	37	179	53	188
PTHash-v1		24		46		67		72		72	
PTHash-v2		38		64		130		155		175	
BBHash-v1		42		118		170		175		175	
BBHash-v2		42		125		180		190		190	

LPHash — Tools and details

- Let’s now quickly see how to achieve this.

Minimizers and super-k-mers

- **Random minimizer.** [Roberts et al., 2004] Given a k-mer x and a random hash function h, the minimizer of x is any m-mer μ such that $h(\mu) \leq h(y)$ for any other m-mer y of x, for some $m < k$.

- **Super-k-mer.** [Li et al., 2013] Given a string s, a super-k-mer g of s is a maximal sub-string of s where each k-mer has the same minimizer μ and μ appears only once in s.

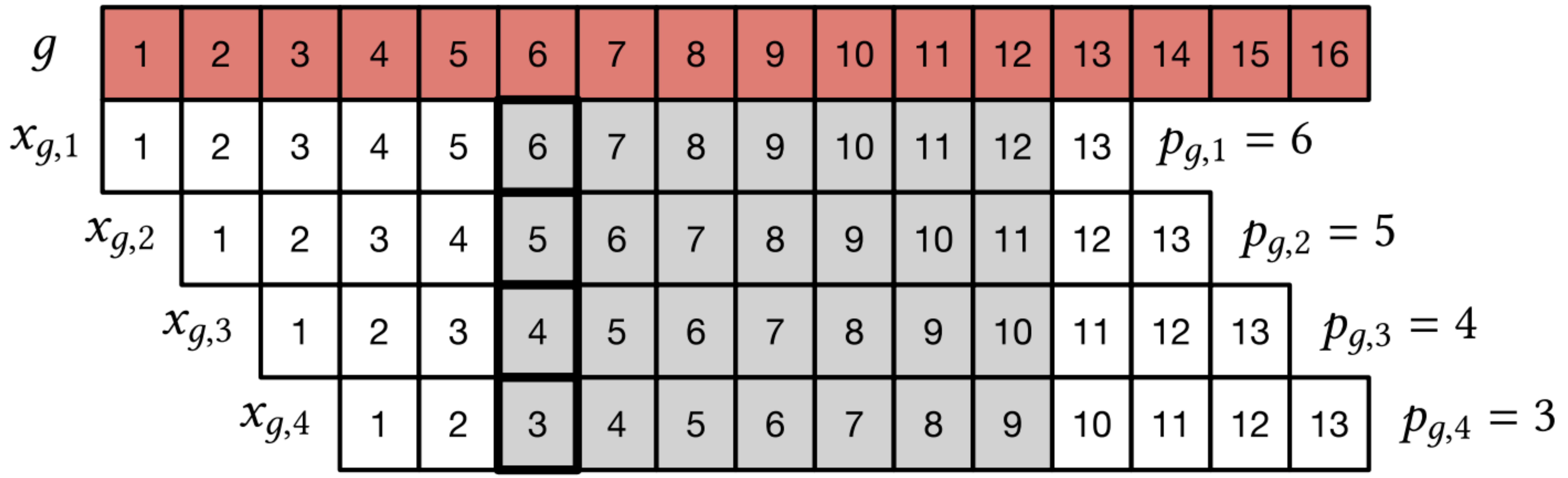
Example for $k = 13$ and $m = 4$:

```
ACGGTAGAACCGATTCAAATTCGATCGATTAATTAGAGCGATAAC...
ACGGTAGAACCGA
 CGGTAGAACCGAT
  GGTAGAACCGATT
   GTAGAACCGATTC
    TAGAACCGATTCA
     AGAACCGATTCAA
      GAACCGATTCAAA
       AACCGATTCAAAT
        ...
```

super-k-mer

Implicitly ranking k-mers with minimizers

- Let g be a super-k-mer and $x_{g,1}, \ldots, x_{g,|g|-k+1}$ its k-mers. Let $p_{g,i}$ be the (starting) position of the minimizer in the i-th kmer of g, $x_{g,i}$.
- Then, Rank$(x_{g,i}) = p_{g,1} - p_{g,i} + 1$.
- Since k-mers are consecutive, the minimizer position "slides" by one position to the left: **rank values are consecutive too → locality is preserved.**



Rank$(x_{g,1}) = 1$

Rank$(x_{g,2}) = 2$

Rank$(x_{g,3}) = 3$

Rank$(x_{g,4}) = 4$

Basic construction

- Given Rank$(x_{g,i}) = p_{g,1} - p_{g,i} + 1$, the idea is to split the evaluation of f in two parts:

$$f(x_{g,i}) = f(x_{g,1}) + \text{Rank}(x_{g,i}) - 1 = f(x_{g,1}) + p_{g,1} - p_{g,i}$$

  for every super-k-mer g.

- This can be implemented using two arrays, L and P.

Basic construction

- Given $\text{Rank}(x_{g,i}) = p_{g,1} - p_{g,i} + 1$, the idea is to split the evaluation of f in two parts:

$$f(x_{g,i}) = \underset{\substack{\uparrow \\ \textbf{global} \\ \text{component}}}{\boxed{f(x_{g,1})}} + \underset{\substack{\uparrow \\ \textbf{local} \\ \text{component}}}{\boxed{\text{Rank}(x_{g,i})}} - 1 = f(x_{g,1}) + p_{g,1} - p_{g,i}$$

for every super-k-mer g.

- This can be implemented using two arrays, L and P.

Basic construction

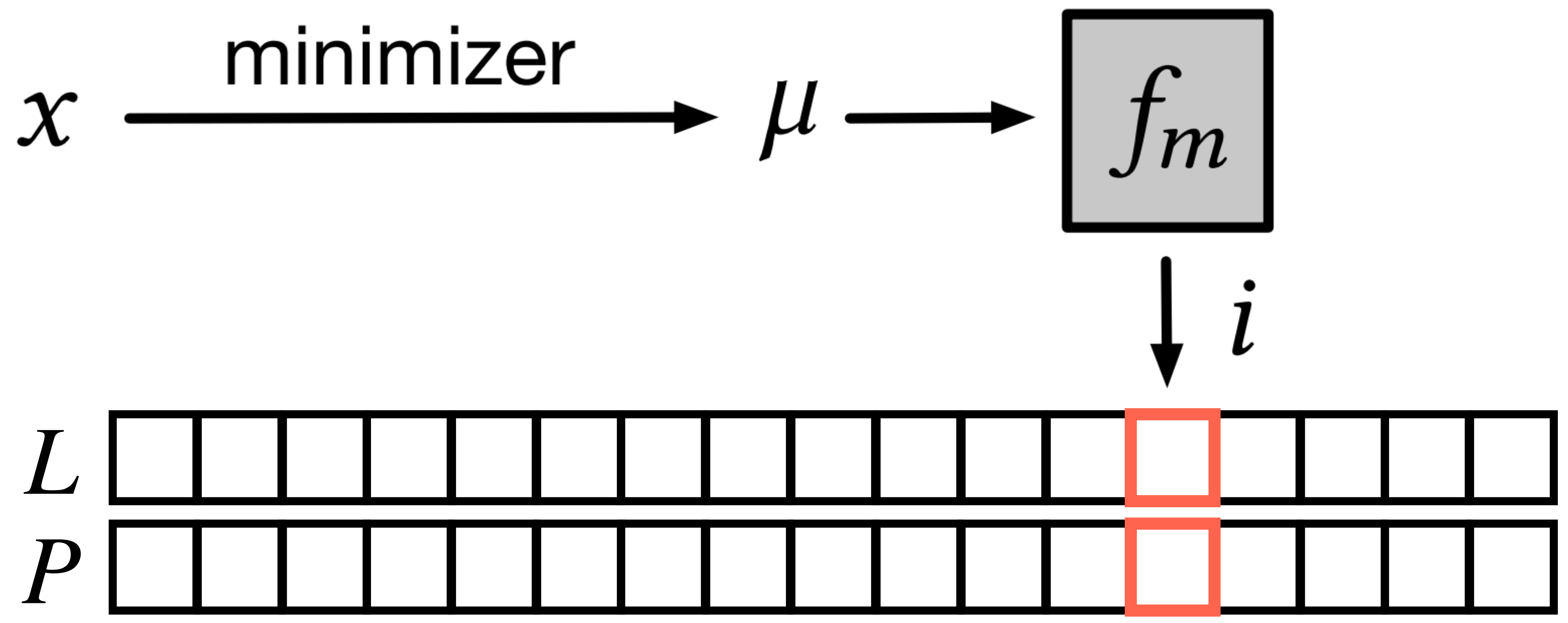
- Given $\text{Rank}(x_{g,i}) = p_{g,1} - p_{g,i} + 1$, the idea is to split the evaluation of f in two parts:

$$f(x_{g,i}) = \underset{\substack{\uparrow \\ \textbf{global} \\ \text{component}}}{\boxed{f(x_{g,1})}} + \underset{\substack{\uparrow \\ \textbf{local} \\ \text{component}}}{\boxed{\text{Rank}(x_{g,i})}} - 1 = \underset{\substack{\uparrow \\ \text{stored} \\ \text{explicitly}}}{\boxed{f(x_{g,1}) + p_{g,1}}} - \underset{\substack{\uparrow \\ \text{computed on-the-fly} \\ \text{at query time}}}{\boxed{p_{g,i}}}$$

for every super-k-mer g.

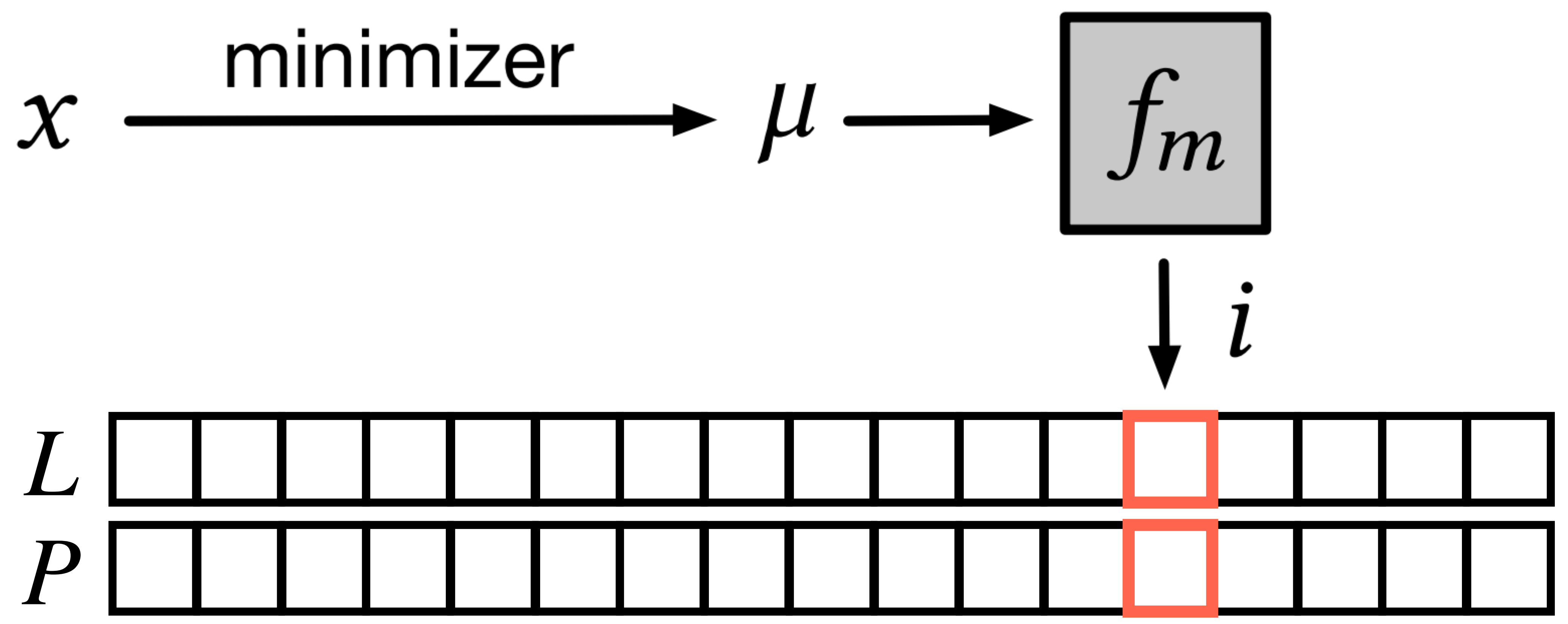
- This can be implemented using two arrays, L and P.

Basic construction

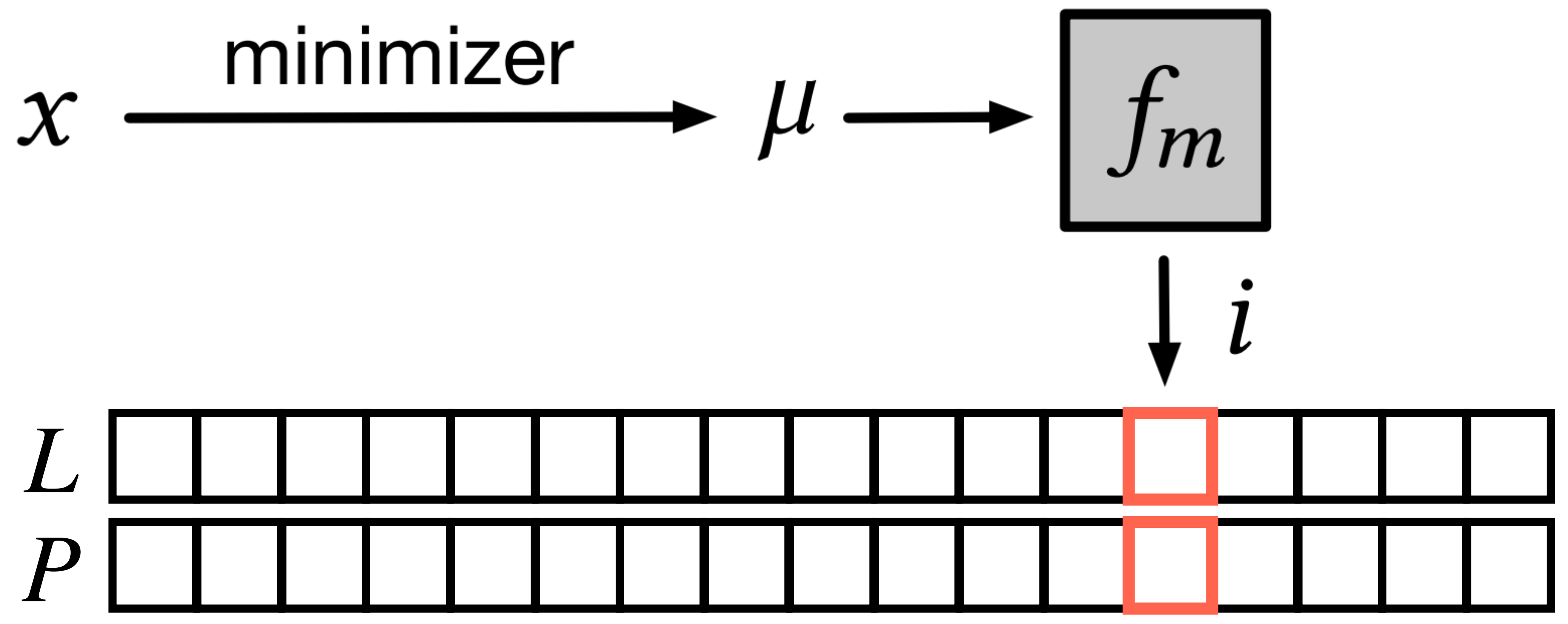


- Where f_m is a MPHF for the set of all the distinct minimizers of the input S.
  For a super-k-mer g whose minimizer μ is such that $f_m(\mu) = i$, let:
  - $L[i]$ be the number of k-mers belonging to super-k-mers having minimizer z such that $f_m(z) < i$. It follows that $f(x_{g,1}) = L[i]$.
  - $P[i] = p_{g,1}$.

Basic construction — Remarks

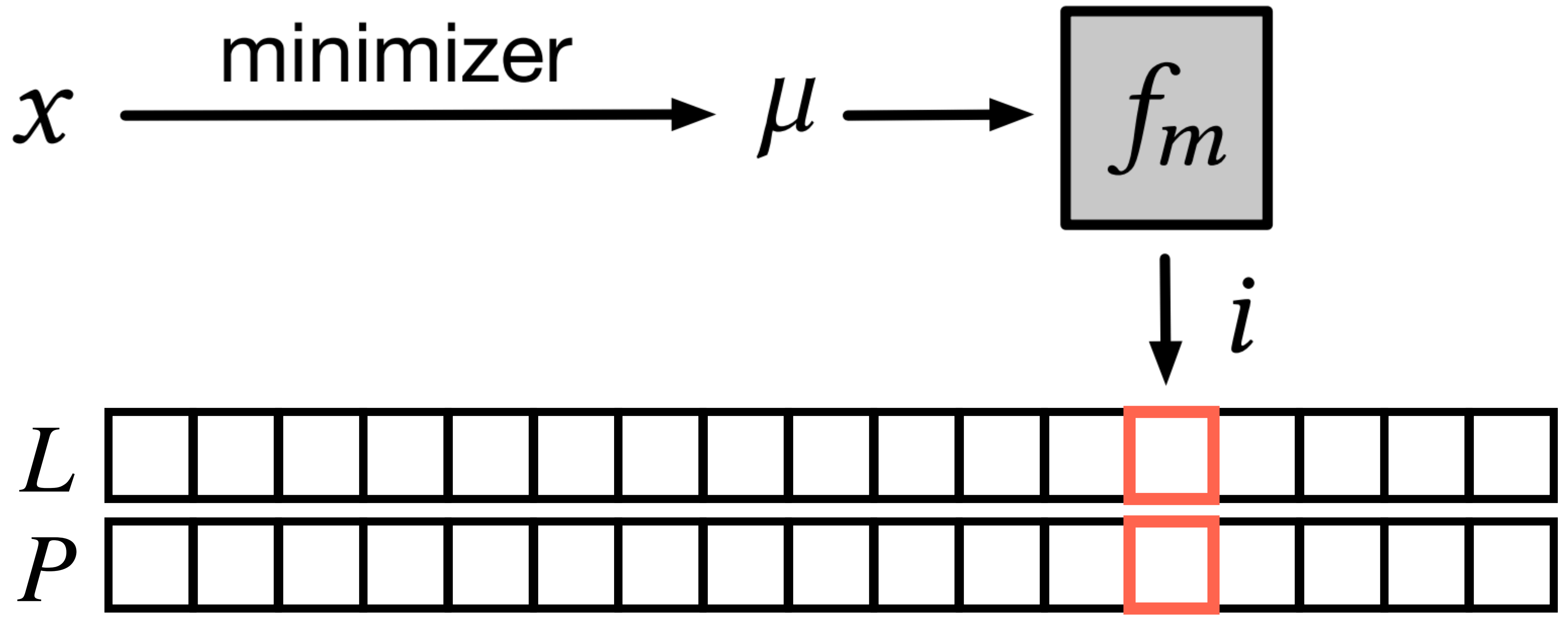


Basic construction — Remarks



1. We compute f by evaluating f_m and with two array accesses.

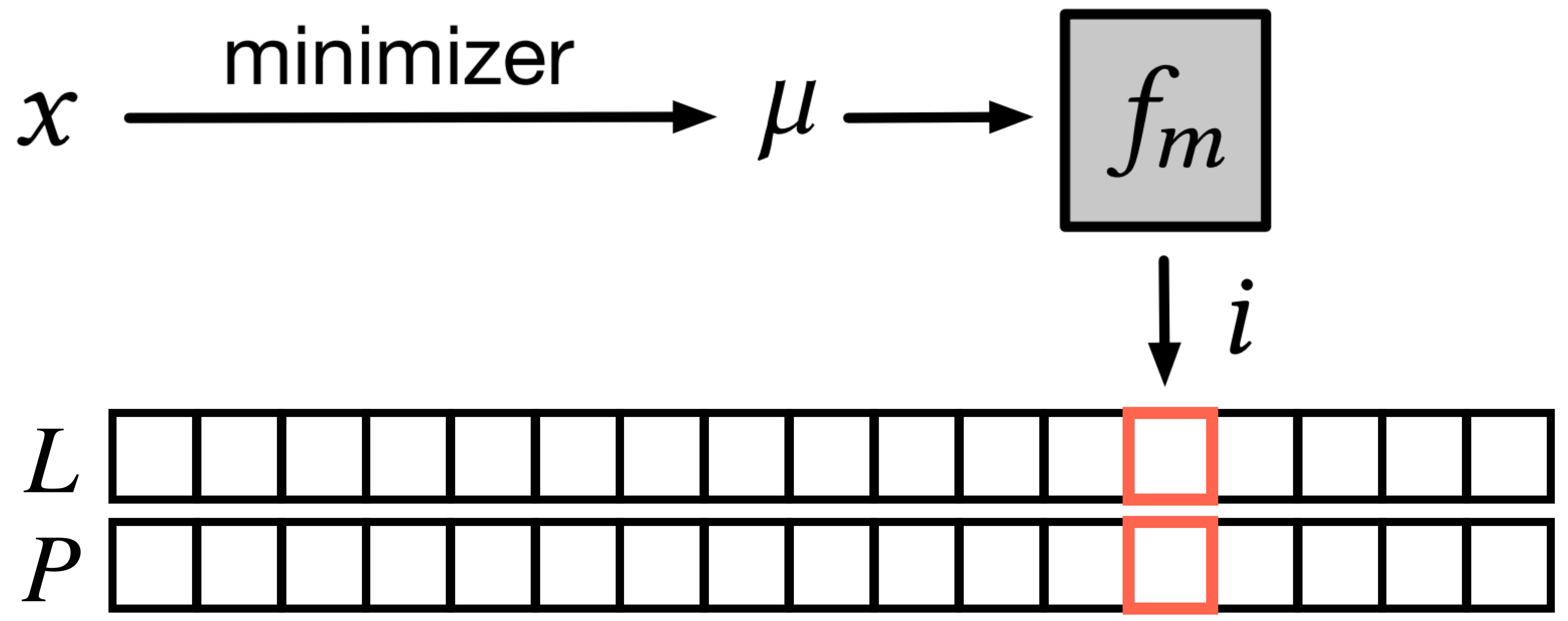
Basic construction — Remarks



1. We compute f by evaluating f_m and with two array accesses.

2. If two consecutive k-mers have the same minimizer: we have already computed $f_m(\mu)$ and accessed L and P, hence we just need to compute the position of the minimizer in the query k-mer (no array accesses nor hash calculations) $\rightarrow$ **faster streaming** queries.

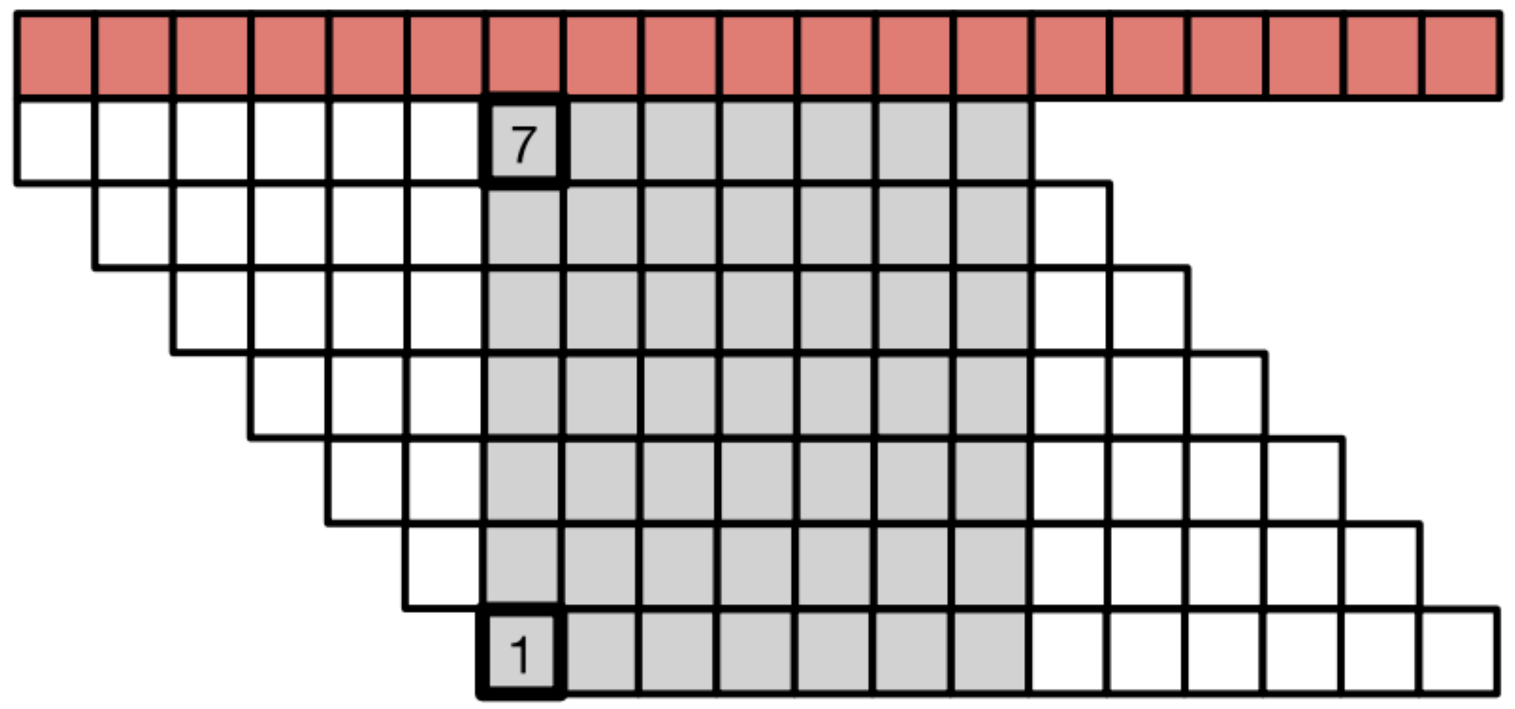
Basic construction — Remarks



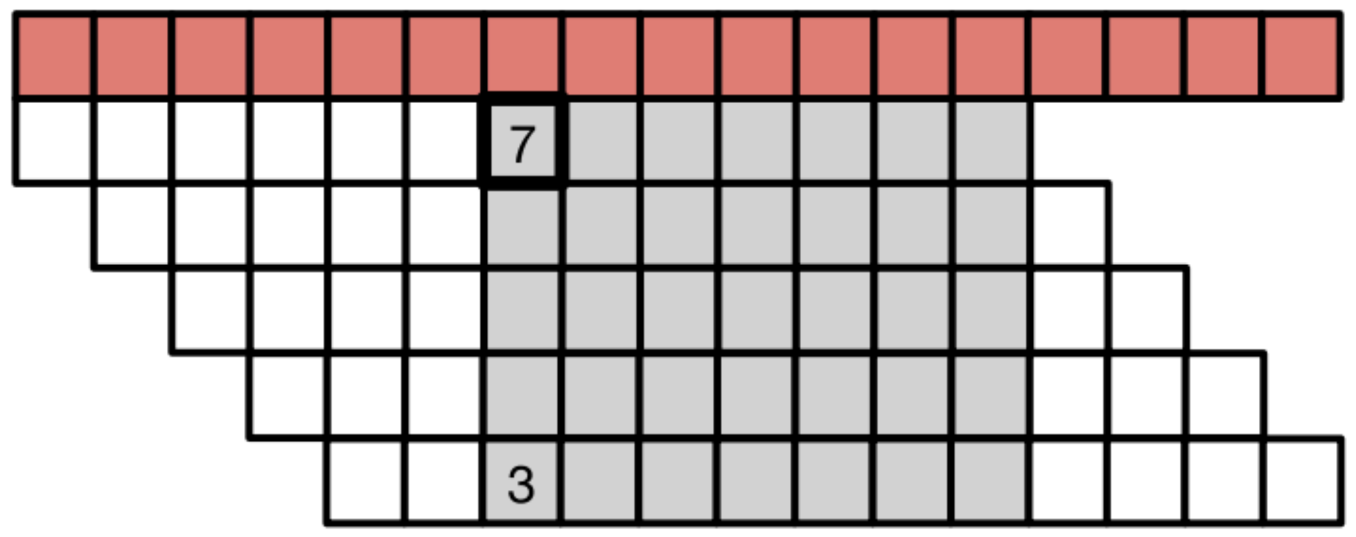
1. We compute f by evaluating f_m and with two array accesses.
2. If two consecutive k-mers have the same minimizer: we have already computed $f_m(\mu)$ and accessed L and P, hence we just need to compute the position of the minimizer in the query k-mer (no array accesses nor hash calculations) $\rightarrow$ **faster streaming** queries.
3. We spend space proportional to the number of minimizers. The expected number of minimizers of length m is $2n/(k-m+2)$. Hence, the **space decreases** when k increases and m is fixed.

Super-k-mer types

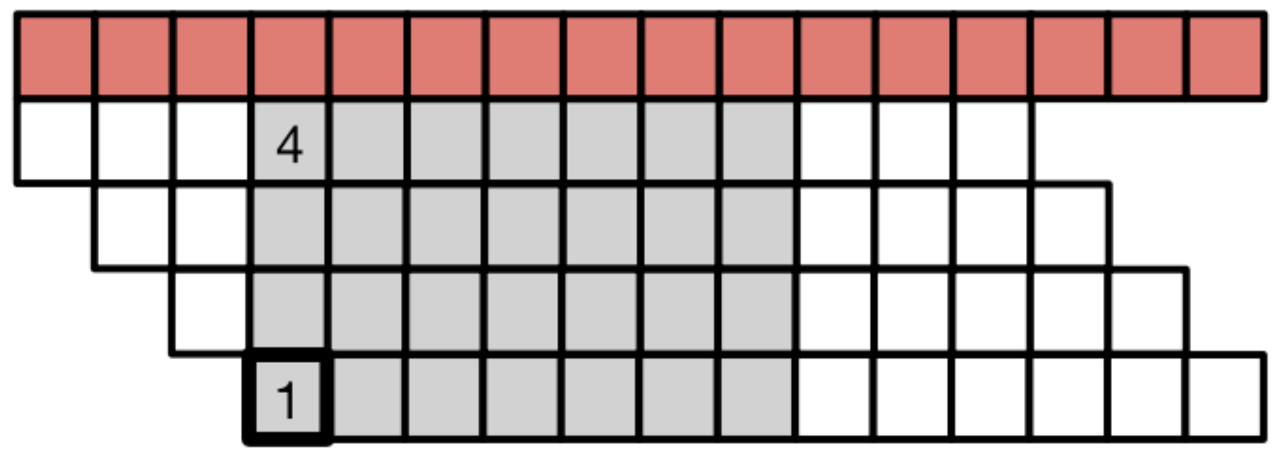
- **FL rule.** Let g be a super-k-mer. Depending on the **position** of the minimizer in its **first** and **last** k-mer, g can be of one of the following *four types*.



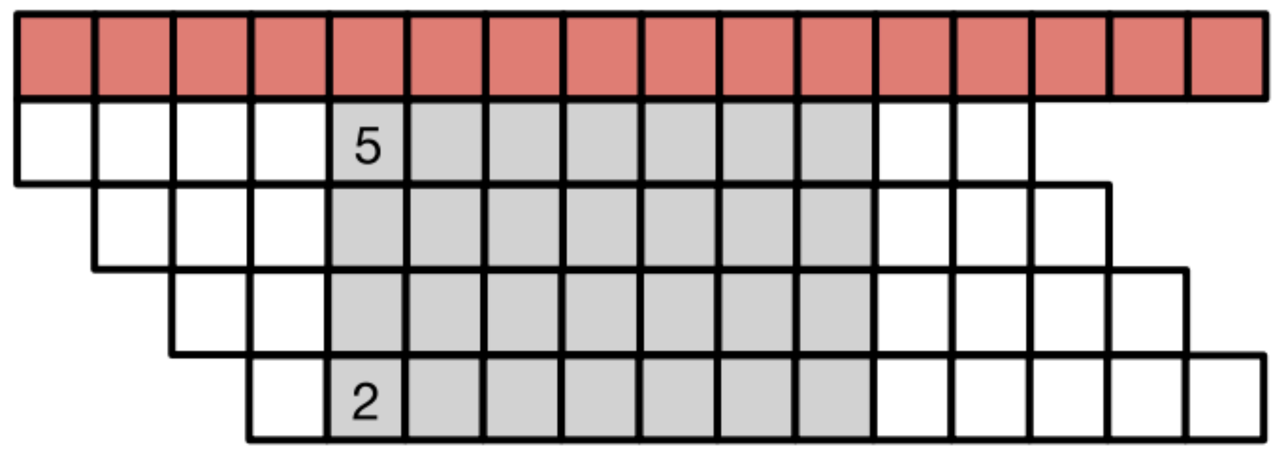
(a) left-right-max



(b) right-max

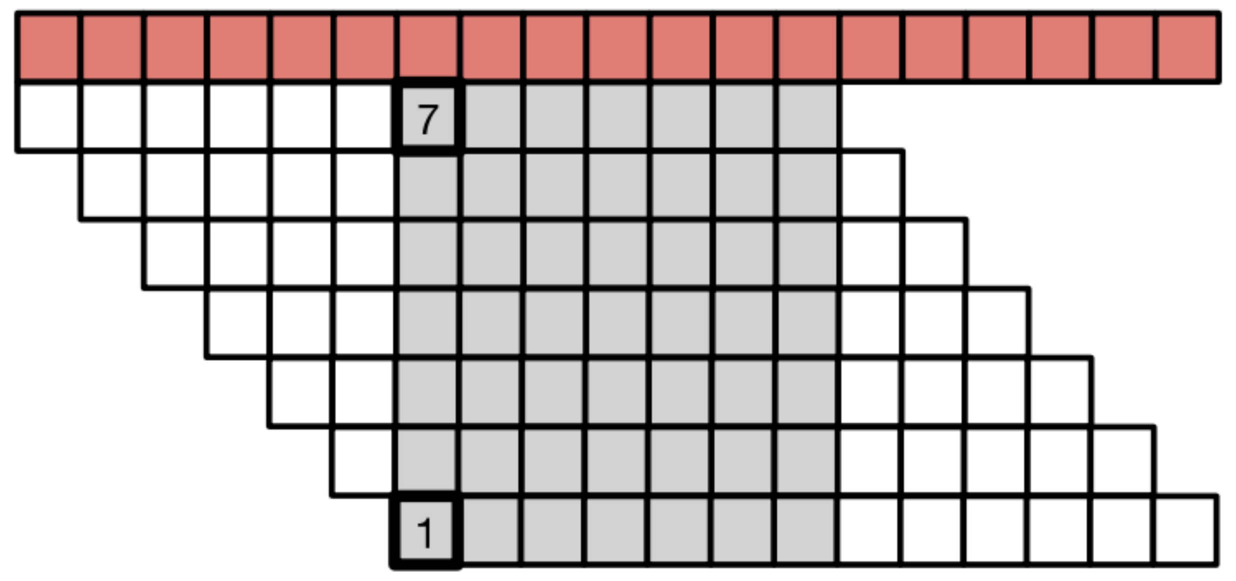


(c) left-max



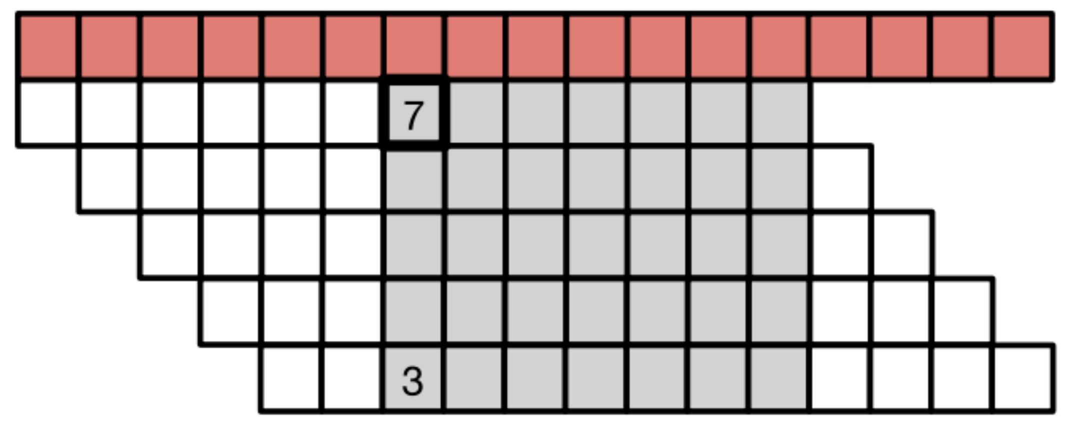
(d) non-max

Super-k-mer types



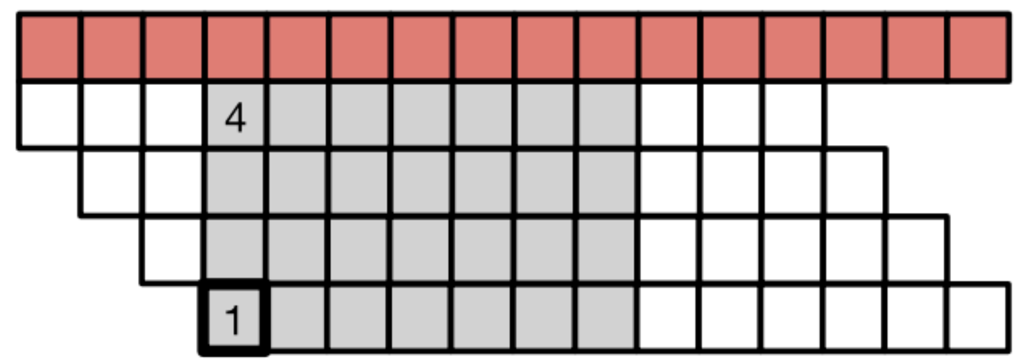
(a) left-right-max

$p_{g,1} = k - m + 1$ and $|g| = k - m + 1 \rightarrow$ no need to store any entry in L and P for *left-right-max* super-k-mers



(b) right-max

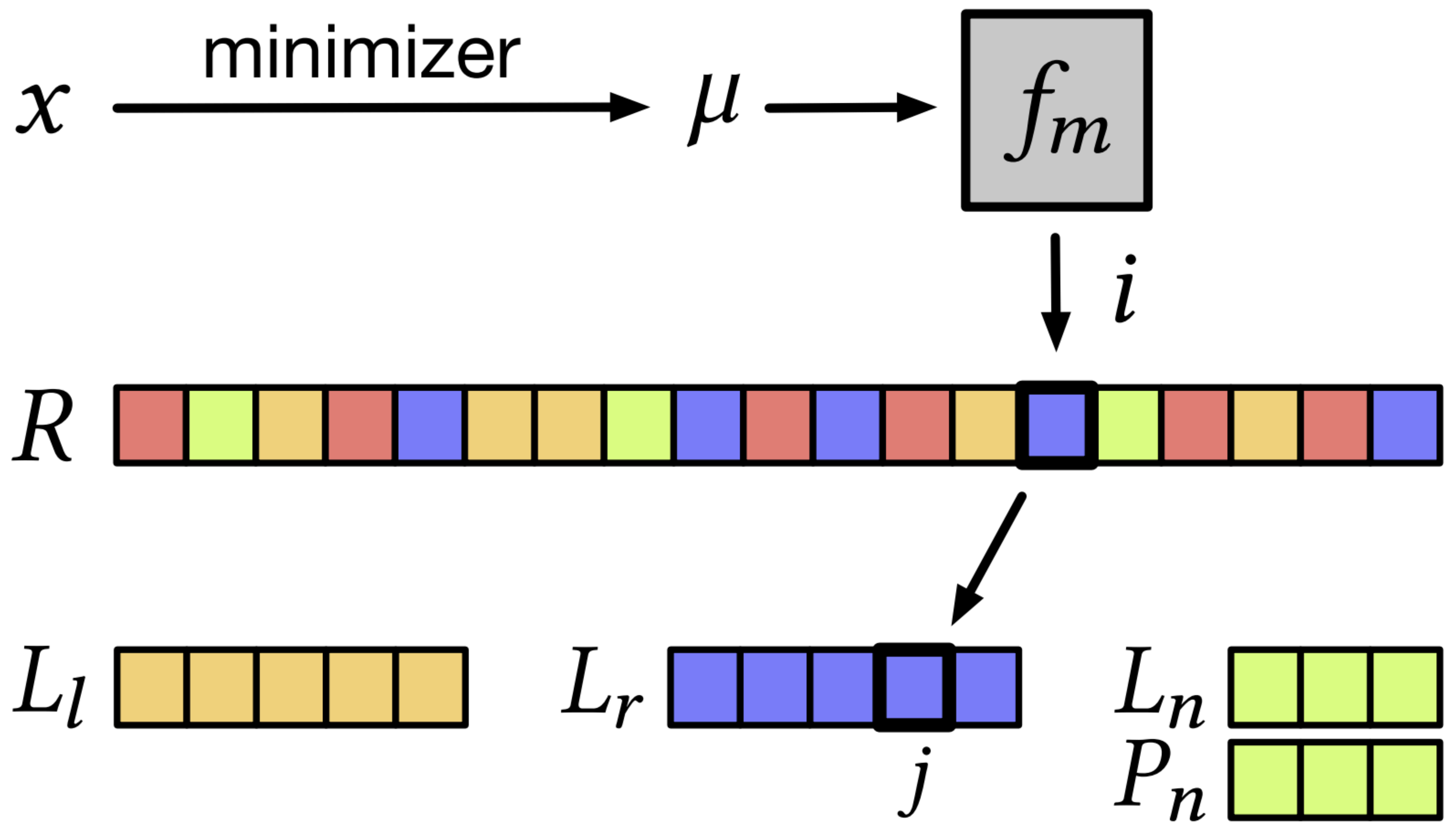
$p_{g,1} = k - m + 1 \rightarrow$ no need to store any entry in P for *right-max* super-k-mers (only store an array L_r)



(c) left-max

$p_{g,1} = |g| - k + 1 \rightarrow$ no need to store any entry in P for *left-max* super-k-mers (only store an array L_l)

Partitioned construction



- Where R is an 2-bit integer array holding the super-k-mer types.

  We need the operation Rank$_t(i)$ for a position i in R and a super-k-mer type t. So, R is represented with a *wavelet tree* [Grossi et al., 2003].

Partitioned construction

- **Q.** Is this effective? Any guarantee?

Partitioned construction

- **Q.** Is this effective? Any guarantee?
- **A.** Yes, for **Theorem 2** below. So for sufficiently large $k - m + 1$, the expected fraction of super-k-mer types will all be $\approx 1/4$, hence we save a lot of space.

Theorem 2. *For any random minimizer scheme* (k, m, h) *we have*

$$P_{lr} = \mathbb{P}[g \text{ is left-right-max}] = W^2 + 1/w$$

$$P_l = \mathbb{P}[g \text{ is left-max}] = W(1 - W)$$

$$P_r = \mathbb{P}[g \text{ is right-max}] = W(1 - W)$$

$$P_n = \mathbb{P}[g \text{ is non-max}] = W^2$$

where $W = \frac{1}{2} \cdot (1 - \frac{1}{w})$ *and* $w = k - m + 1$.

Ambiguous minimizers

- If a minimizer appears in two or more super-k-mers, we say it is *ambiguous*.
- In this case, a single minimizer position is **not** enough to rank k-mers without ambiguity.
- We therefore build a separate MPHF for all k-mers belonging to super-k-mers having ambiguous minimizers.
- The fraction of ambiguous minimizers is **small**, e.g., 1—4% on the datasets we tested in the paper.

Conclusions

- LPHash is an efficient solution to the minimal perfect hashing problem for k-mer sets.
- Example: **0.87 bits/k-mer** on the human genome (2.7B k-mers, for k=63) with very fast streaming queries.
- Space usage decreases for increasing k, and in the (near) future we are going to have longer k-mers.
- Better solutions to classic problems if we restrict our attention to specific input classes.
- LPHash ingredients:

  - implicit ranking of k-mers with minimizers;

  - structural characterisation of super-k-mers.

Thank you for the attention!