

# Sparse and Skew Hashing of K-Mers\*

\*ISMB 2022, to appear.

## Giulio Ermanno Pibiri

ISTI-CNR, [giulio.ermanno.pibiri@isti.cnr.it](mailto:giulio.ermanno.pibiri@isti.cnr.it)



@giulio\_pibiri



@jermmp

**RECOMB-seq** (RECOMB Satellite Workshop on Massively Parallel Sequencing)  
La Jolla, USA, 20-21 May 2022

# The K-Mer Dictionary Problem

- **K-Mer.** A  $k$ -mer is a string of length  $k$  over the DNA alphabet  $\{A,C,G,T\}$ .
- We are given a large DNA string (e.g., a genome or a pan-genome) and let  $K$  be the set of all its  $n$  distinct  $k$ -mers.

Example: The human genome (GRCh38) has  $>2.5B$  distinct  $k$ -mers for  $k = 31$ .

- **Problem.** We want to build a dictionary for  $K$  so that the following operations are efficient:
  - $i = \text{Lookup}(g)$ , where  $0 \leq i < n$  if the  $k$ -mer  $g \in K$  or  $i = -1$  otherwise;
  - return the  $k$ -mer  $g = \text{Access}(i)$  if  $0 \leq i < n$ .

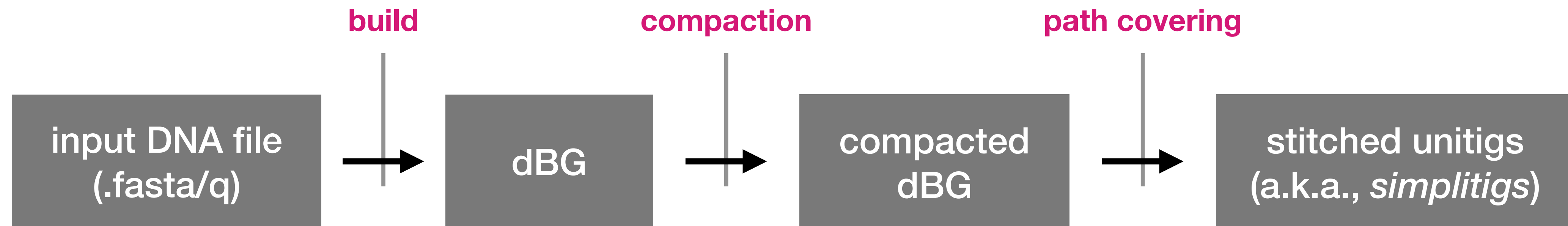
(Other operations of interest are *iteration* and *streaming* membership queries. See the paper for details.)

# K-Mer Applications

- Software tools based on  $k$ -mers are predominant in Bioinformatics.
- Many applications:
  - genome assembly
  - variant calling
  - pan-genome analysis
  - meta-genomics
  - sequence comparison/alignment
  - ...

# de Bruijn Graphs

- **Fact.** Equivalence between a set of  $k$ -mers and a *de Bruijn* graph (dBG).
- There are efficient software tools to run the following pre-processing flow.



- BCALM [Chikhi et al., 2016]
- Cuttlefish [Khan and Patro, 2021]

- A collection of DNA strings with **no duplicate**  $k$ -mers.
- Efficient heuristic method to reduce the number of bases, e.g, UST [Rahman and Medvedev, 2020].

# Super-k-Mers

- **Property.** Consecutive  $k$ -mers are likely to have the same *minimizer* [Roberts et al., 2004] — the smallest sub-string of length  $m \leq k$  according to a given order  $R$ .

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

ACGGTAG**AACC**GATTCAAATTCGATCGATTAATTAGAGCGATAAC...

ACGGTAG**AACC**GA

CGGTAG**AACC**GAT

GGTAG**AACC**GATT

GTAG**AACC**GATTC

TAG**AACC**GATTCA

AG**AACC**GATTCAA

G**AACC**GATTCAAA

AACCGATTCA**AAAT**

...

super- $k$ -mer

- **Super-k-mer.** [Li et al., 2013] Given a string, a *super-k-mer* is a *maximal* sequence of consecutive  $k$ -mers having the same minimizer.

# Super-k-Mers

- **Observation 1.** Since consecutive  $k$ -mers are likely to have the same minimizers, there are *far fewer* super- $k$ -mers than  $k$ -mers — approx.  $(k - m + 2)/2$  times less for *random* minimizers — → **sparse** indexing.
- **Observation 2.** A super- $k$ -mer of length  $s$  is a **space-efficient** representation of the set of its constituent  $s - k + 1$   $k$ -mers:  $2s/(s - k + 1)$  vs.  $2k$  bits/ $k$ -mer. If  $s$  is sufficiently large and/or we have long chains of super- $k$ -mers, the cost becomes approx. 2 bits/ $k$ -mer.

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

ACGGTAGAACCGATTCAAATTCGATCGATTAATT...

ACGGTAGAACCGATTCAAATTCGATCGATTAATT...



This **chain** is of length 31 and costs  $2 \times 31 = 62$  bits for 19  $k$ -mers (3.26 bits/ $k$ -mer).

This super- $k$ -mer costs  $2 \times 19 = 38$  bits for 7  $k$ -mers  
(5.43 bits/ $k$ -mer vs.  $2 \times 13 = 26$  bits/ $k$ -mer).

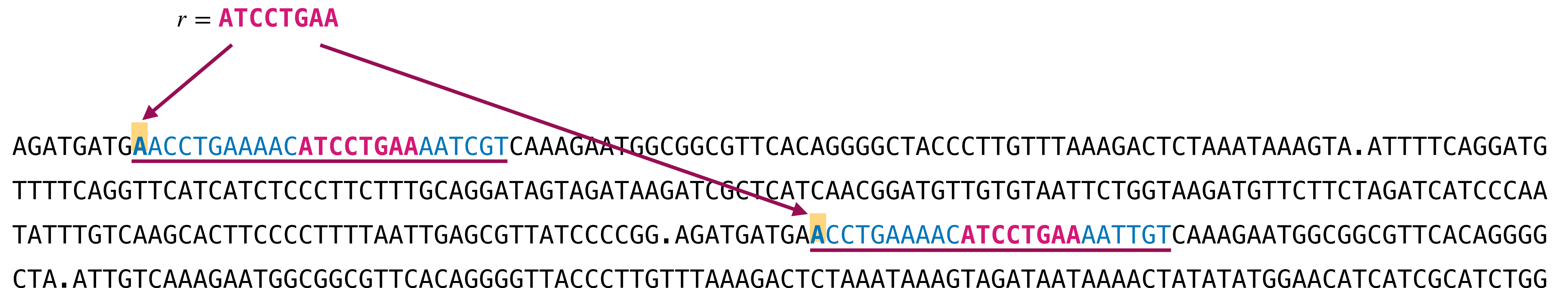


ACGGTAGAACCGATTCAAATTCGATCGATTAATT...  $s=19$

AACCGATTCAAATTCGATCGATTAATT...  $s=24$

# Sparse Hashing

- **Q.** How to index super- $k$ -mers?
- Do **not** break the chains of super- $k$ -mers to avoid wasting  $2(k - 1)$  bits per super- $k$ -mer.
- Locate super- $k$ -mers with an array of offsets into the strings, indexed by a **minimal perfect hash function** (MPHF) on the minimizers.
- Upon  $\text{Lookup}(g)$ : if  $r$  is the minimizer of  $g$ , locate and scan the “bucket” of  $r$  — the set of super- $k$ -mers that have minimizer  $r$ .





# Skew Hashing

- **Problem.** Some buckets can be very large.

For example on the human genome (GRCh38), for  $k = 31$  and  $m = 20$ : largest bucket size can be as large as  $3.6 \times 10^4$ .

- **Property.** Minimizers have a (very) **skew** distribution for sufficiently long length.

Bucket size distribution (%) for  $k = 31$  and the first  $n = 10^9$   $k$ -mers of the human genome, by varying minimizer length  $m$ .

size / $m$	11	12	13	14	15	16	17	18	19	20	21
1	13.7	19.8	29.7	42.4	61.5	79.5	89.8	94.4	96.3	97.1	97.5
2	7.5	10.6	14.4	17.7	19.4	13.6	7.3	3.9	2.4	1.7	1.4
3	5.2	7.3	8.8	10.4	8.4	3.7	1.4	0.8	0.5	0.4	0.4
4	4.0	5.5	6.0	7.0	4.1	1.3	0.5	0.3	0.2	0.2	0.2
5	3.2	4.4	4.5	5.0	2.2	0.6	0.3	0.2	0.1	0.1	0.1

On the **full** human genome (GRCh38), for  $k = 31$  and  $m = 20$ :

2,505,445,761  $k$ -mers

421,845,806 minimizers

388,018,280 (91.98%) only appear **once!**



# Skew Hashing

- We fix an integer  $\ell$ : by virtue of the skew distribution, the fraction of buckets having **more than**  $2^\ell$  super- $k$ -mers is **small**.
- So, we can afford a MPHf over the set of  $k$ -mers that belong to such super- $k$ -mers. The output of the MPHf for a  $k$ -mer  $g$  is the **identifier** of the super- $k$ -mer where  $g$  is present. (This identifier can be written in few bits; see the paper for details.)
- Upon Lookup, we will scan **one** super- $k$ -mer only.

Bucket size distribution (%) for  $k = 31$  and the first  $n = 10^9$   $k$ -mers of the human genome, by varying minimizer length  $m$ .

size / $m$	11	12	13	14	15	16	17	18	19	20	21
1	13.7	19.8	29.7	42.4	61.5	79.5	89.8	94.4	96.3	97.1	97.5
2	7.5	10.6	14.4	17.7	19.4	13.6	7.3	3.9	2.4	1.7	1.4
3	5.2	7.3	8.8	10.4	8.4	3.7	1.4	0.8	0.5	0.4	0.4
4	4.0	5.5	6.0	7.0	4.1	1.3	0.5	0.3	0.2	0.2	0.2
5	3.2	4.4	4.5	5.0	2.2	0.6	0.3	0.2	0.1	0.1	0.1

For  $\ell = 2$ , just  
 $100.0 - (97.1 + 1.7 + 0.4 + 0.2)\% = 0.6\%$  of  
buckets with more than  $2^{\ell=2} = 4$  super- $k$ -mers.

# Benchmarking SSHash

- Code in **C++17**, compiled with flags: `-O3 -march=native`.
- All experiments are **single-threaded**.
- We use  $\ell = 6$  for all experiments.
- We use  $m = 17, 17, 20, 20$  for respectively Cod, Kestrel, Human, and Bacterial.  
(A good rule of thumb is  $m = \lceil \log_4(N) \rceil + 1$  or  $m = \lceil \log_4(N) \rceil + 2$ .)

Some basic statistics for the datasets used in the experiments, for  $k = 31$ , such as number of:  $k$ -mers ( $n$ ), paths ( $p$ ), and bases ( $N$ ).

Dataset	$n$	$p$	$N$	$\lceil \log_2(N) \rceil$
Cod	502,465,200	2,406,681	574,665,630	30
Kestrel	1,150,399,205	682,344	1,170,869,525	31
Human	2,505,445,761	13,014,641	2,895,884,991	32
Bacterial	5,350,807,438	26,449,008	6,144,277,678	33

**NOTE:** We used BCALM (v2) [Chikhi et al., 2016] to build the compacted DBG and then UST [Rahman and Medvedev, 2020] to compute the stitched unitigs.

# Competitors

- **DBG-FM** [Chikhi et al., 2014]: FM-index [Ferragina and Manzini, 2000]
- **Pufferfish** [Almodaresi et al., 2018]: MPHF
- **Blight** [Marchet et al., 2021]: MPHF+minimizers

# Overall Comparison — Space and Lookup

Dictionary space in total GB and average bits/ $k$ -mer (bpk).

Dictionary	Cod		Kestrel		Human		Bacterial	
	GB	bpk	GB	bpk	GB	bpk	GB	bpk
dBG-FM, $s = 128$	0.22	3.48	0.44	3.07	–	–	–	–
dBG-FM, $s = 64$	0.27	4.38	0.55	3.86	–	–	–	–
dBG-FM, $s = 32$	0.39	6.16	0.78	5.43	–	–	–	–
Pufferfish, sparse	1.75	27.80	3.69	25.66	8.87	28.32	18.91	28.28
	1.49	23.70	3.37	23.40	7.50	23.96	16.09	24.06
Pufferfish, dense	2.69	42.76	5.97	41.54	14.11	45.04	30.70	45.89
	2.43	38.66	5.65	39.28	12.74	40.68	27.88	41.68
Blight, $b = 4$	0.91	14.53	2.16	15.00	5.04	16.11	11.40	17.04
Blight, $b = 2$	1.04	16.57	2.45	17.04	5.67	18.13	12.74	19.05
Blight, $b = 0$	1.17	18.61	2.74	19.06	6.32	20.17	14.12	21.11
SSHash, regular	0.44	6.98	0.93	6.48	2.59	8.28	5.50	8.22
SSHash, canonical	0.50	7.92	1.00	7.30	2.94	9.39	6.17	9.22

Dictionary Lookup time in average ns/ $k$ -mer.

Dictionary	Cod		Kestrel		Human		Bacterial	
	Lkp <sup>+</sup>	Lkp <sup>-</sup>	Lkp <sup>+</sup>	Lkp <sup>-</sup>	Lkp <sup>+</sup>	Lkp <sup>-</sup>	Lkp <sup>+</sup>	Lkp <sup>-</sup>
dBG-FM, $s = 128$	22,980	16,501	23,934	16,764	–	–	–	–
dBG-FM, $s = 64$	15,013	10,919	15,929	11,462	–	–	–	–
dBG-FM, $s = 32$	11,386	7,929	11,703	8,073	–	–	–	–
Pufferfish, sparse	1110	700	5456	769	13,656	862	27,748	983
	624	439	635	485	720	519	816	582
Pufferfish, dense	624	439	635	485	720	519	816	582
	624	439	635	485	720	519	816	582
Blight, $b = 4$	2520	2751	2743	3104	2820	3329	3105	3913
Blight, $b = 2$	1800	1643	1916	1820	2008	1975	2095	2146
Blight, $b = 0$	1571	1317	1692	1472	1780	1610	1859	1751
SSHash, regular	1045	1158	1042	1265	1338	1530	1389	1780
SSHash, canonical	834	690	882	781	990	854	1051	995

- Compared to BWT-based indexes: **one order of magnitude faster** for “just” 2x more space.
- Compared to other hashing schemes: **2-5x smaller** with comparable of faster query time.



# Overall Comparison — Streaming Queries

Query time for streaming membership queries for various dictionaries. The query time is reported as total time in minutes (tot), and average ns/ $k$ -mer (avg). We also indicate the query file (SRR number) and the percentage of hits. Both high-hit ( $> 70\%$  hits) and low-hit ( $< 1\%$  hits) workloads are considered.

Dictionary	Cod		Kestrel		Human		Bacterial	
	SRR12858649		SRR11449743		SRR5833294		SRR5901135	
	81.37% hits		74.60% hits		91.65% hits		87.79% hits	
	tot	avg	tot	avg	tot	avg	tot	avg
Pufferfish, sparse	0.6	214	14.1	609	17.0	651	9.1	691
Pufferfish, dense	0.2	92	8.5	368	10.5	402	5.3	404
Blight, $b = 4$	2.1	766	32.5	1400	27.3	1041	11.4	864
Blight, $b = 2$	1.2	453	16.6	714	17.5	670	8.6	648
Blight, $b = 0$	0.8	282	10.8	464	11.5	440	5.8	434
SSHash, regular	0.5	166	6.2	267	8.2	311	3.0	223
SSHash, canonical	0.3	111	5.1	219	6.7	253	2.4	184

(a) high-hit workload

Dictionary	Cod		Kestrel		Human		Bacterial	
	SRR11449743		SRR12858649		SRR5901135		SRR5833294	
	0.659% hits		0.484% hits		0.002% hits		0.086% hits	
	tot	avg	tot	avg	tot	avg	tot	avg
Pufferfish, sparse	14.6	627	0.9	312	11.3	855	25.5	975
Pufferfish, dense	8.7	374	0.2	92	5.8	435	13.6	518
Blight, $b = 4$	72.2	3112	6.6	2407	35.7	2704	253.2	9675
Blight, $b = 2$	45.9	1978	3.0	1115	19.1	1445	117.7	4498
Blight, $b = 0$	18.1	780	1.8	655	14.4	1088	32.2	1232
SSHash, regular	10.7	463	0.9	314	6.2	463	14.3	544
SSHash, canonical	5.1	220	0.4	155	2.5	183	6.4	244

(b) low-hit workload

# Conclusions

- SShash is an efficient solution to the *K-Mer Dictionary problem*: **good trade-off** between space and time.
- Compared to BWT-based indexes: one order of magnitude faster for “just” 2X more space. Compared to other hashing schemes: 2-5X smaller with comparable or faster query time.
- Tool-box: spectrum-preserving string sets (SPSSs), minimizers, minimal perfect hashing (MPHF, <https://github.com/jermp/pthash>), Elias-Fano.
- Ingredients:
  - **Sparse indexing** to obtain good space effectiveness;
  - **Skew hashing** to guarantee fast lookup for “heavy” buckets.
- Code in C++17 is available at: <https://github.com/jermp/sshash>.

**Thank you for the attention!**



# Preliminary Observations

- The algorithmic literature about (*compressed*) *string dictionaries* is rich of solutions [[Martínez-Prieto et al., 2016](#)] (e.g., Front-Coding, path-decomposed tries, double-array tries), but are relevant for “generic strings”:
  - variable-length,
  - larger alphabets (e.g., ASCII),
  - (usually) no particular properties of the strings to aid compression.
- Since  $k$ -mers are extracted *consecutively* from DNA, a  $k$ -mer following another one shares  $k - 1$  bases (very low entropy).

ACGGTAGAACCGATTCAAATTCGACGTAGC...

A**CGGTAGAACCGA**

**CGGTAGAACCGAT**

GGTAGAACCGATT

GTAGAACCGATT

TAGAACCGATTCA

...



Example for  $k = 13$ .

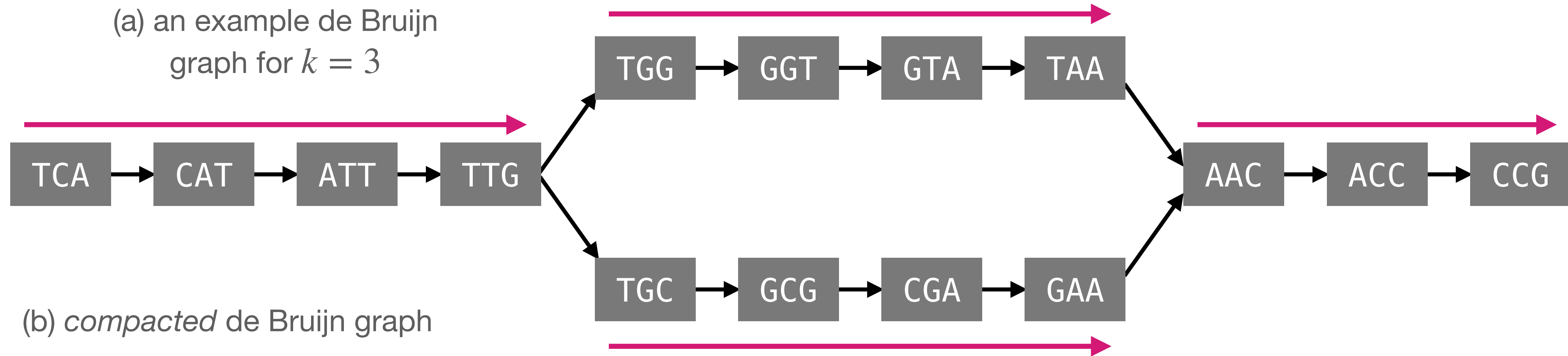
# A World of K-Mer Indexes

- Huge research effort produced many types of indexes based on  $k$ -mers, with different:
  - representations (hashing, BWT-based, exact vs. approximate),
  - features (e.g., static vs. dynamic),
  - space/time trade-offs,
  - operations, ecc.
- Recent surveys on this topic:
  - [Data Structures based on  \$k\$ -mers for Querying Large Collections of Sequencing Data Sets](#)  
Marchet et al., Genome Research, 2020.
  - [Data Structures to Represent a Set of  \$k\$ -long DNA Sequences](#)  
Chikhi et al., ACM Computing Surveys, 2021.

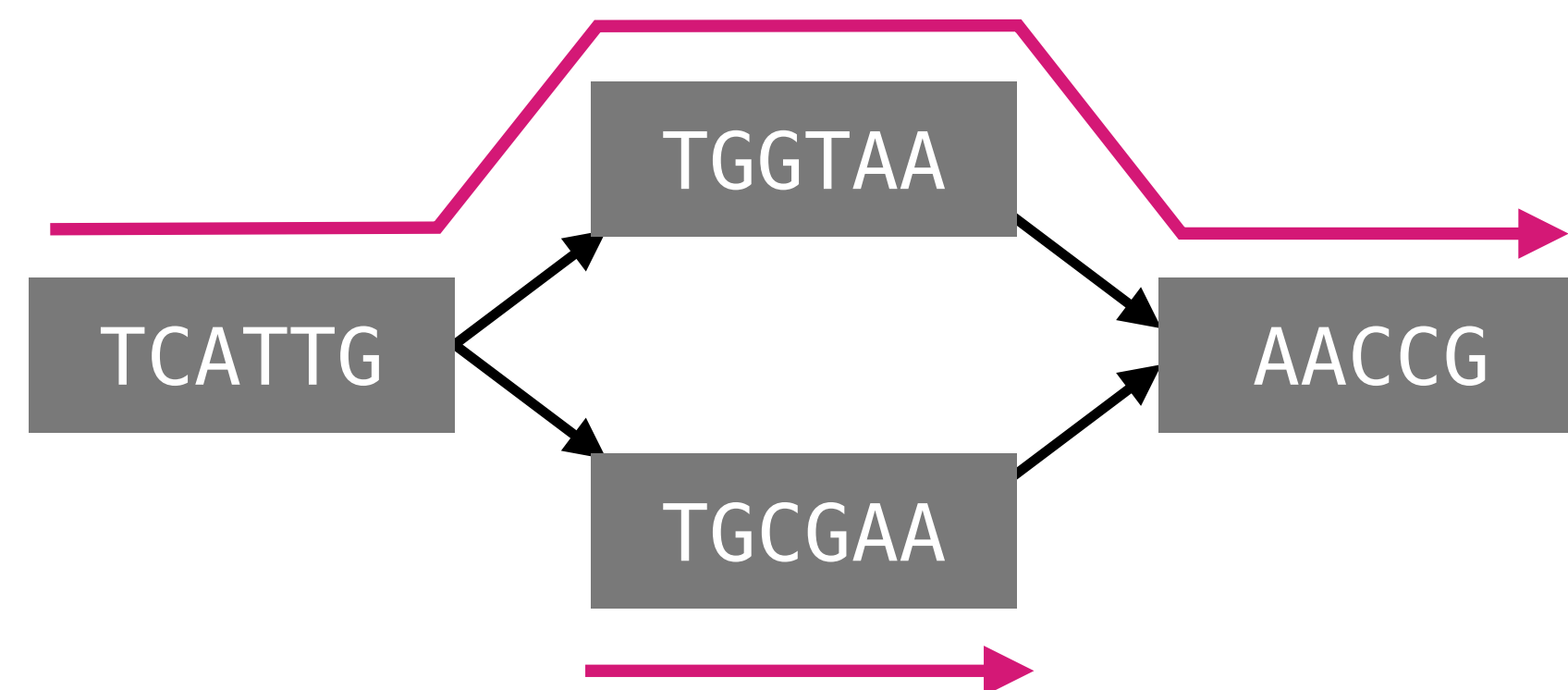
# de Bruijn Graphs

**Fact.** Equivalence between a set of  $k$ -mers and a *de Bruijn* graph (dBG).

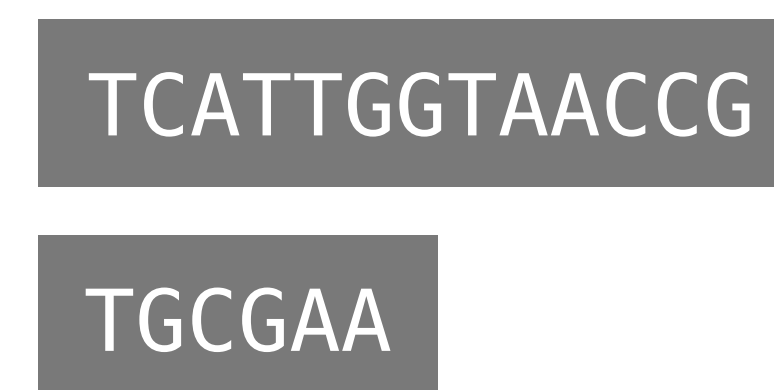
(a) an example de Bruijn graph for  $k = 3$



(b) compacted de Bruijn graph



(c) set of *stitched (maximal) unitigs*



# Minimizers

- **Minimizer.** [Roberts et al., 2004] Given a  $k$ -mer and an order relation  $R$ , the *minimizer* of length  $m \leq k$  is the *smallest*  $m$ -mer of the  $k$ -mer according to  $R$ .
- Example. Given  $g = \text{ACGGTAGAACCGA}$  ( $k = 13$ ) and  $m = 4$ :

ACGG	$h(\text{ACGG}) = 9842978325$	
CGGT	$h(\text{CGGT}) = 817612312$	
GGTA	$h(\text{GGTA}) = 8265731$	← <i>smallest hash code</i>
GTAG	$h(\text{GTAG}) = 478491248$	
TAGA	$h(\text{TAGA}) = 17491411$	
AGAA	$h(\text{AGAA}) = 17148914$	
GAAC	$h(\text{GAAC}) = 91815379$	
<b>AACC</b>	$h(\text{AACC}) = 645793914$	
ACCG	$h(\text{ACCG}) = 918417644$	
CCGA	$h(\text{CCGA}) = 814188124$	

If  $R$  is the *lexicographic* order.

If  $R$  is defined by a random hash function  $h$ .

# Minimal Perfect Hashing

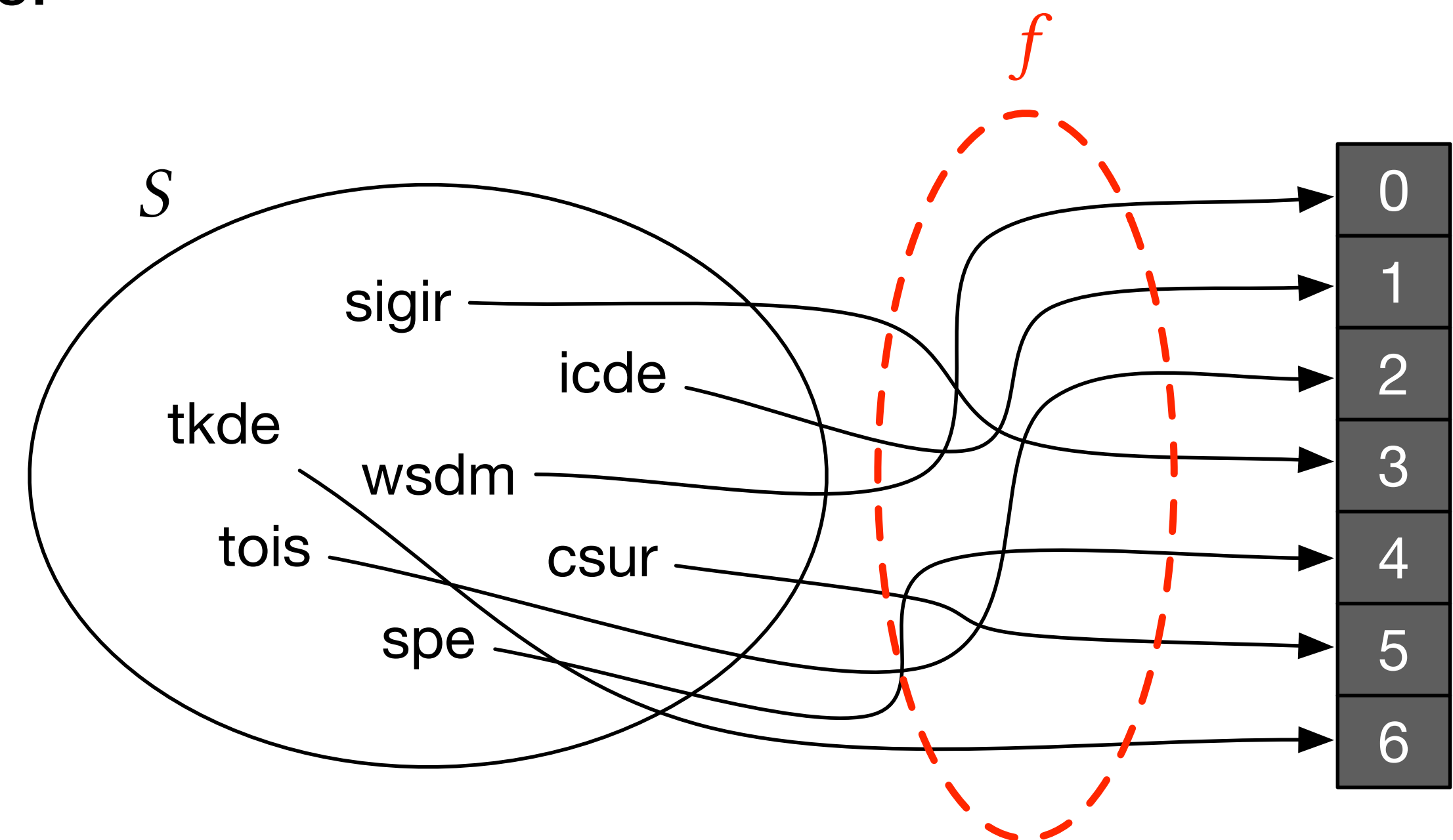
**MPHF.** Given a set  $S$  of  $n$  distinct keys, a function  $f$  that *bijectively* maps the keys of  $S$  into the range  $\{0, \dots, n - 1\}$  is called a *minimal perfect hash function* (MPHF) for  $S$ .

- Lower bound of 1.44 bits/key — in practice: 2-4 bits/key and constant time evaluation.

- Many algorithms available:

- 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]

<https://github.com/jermp/pthash>



# Sparse Hashing — Example

a collection of 4 stitched unitigs:  
285  $k$ -mers for  $k = 31$ ,  $N = 408$  bases



```
AGATGATGAACCTGAAAACATCCTGAAAATCGTCAAAGAATGGCGG
CGTTCACAGGGGCTACCCTTGTTTAAAGACTCTAAATAAAGTA.AT
TTTCAGGATGTTTTTCAGGTTTCATCATCTCCCTTCTTTGCAGGATAG
TAGATAAGATCGCTCATCAACGGATGTTGTGTAATTCTGGTAAGAT
GTTCTTCTAGATCATCCCAATATTTGTCAAGCACTTCCCCTTTTAA
TTGAGCGTTATCCCCGG. AGATGATGAACCTGAAAACATCCTGAAA
ATTGTCAAAGAATGGCGGCGTTCACAGGGGCTA. ATTGTCAAAGAA
TGGCGGCGTTCACAGGGGTTACCCTTGTTTAAAGACTCTAAATAAA
GTAGATAATAAACTATATATGGAACATCATCGCATCTGG
```

24 minimizers, for  $m = 8$



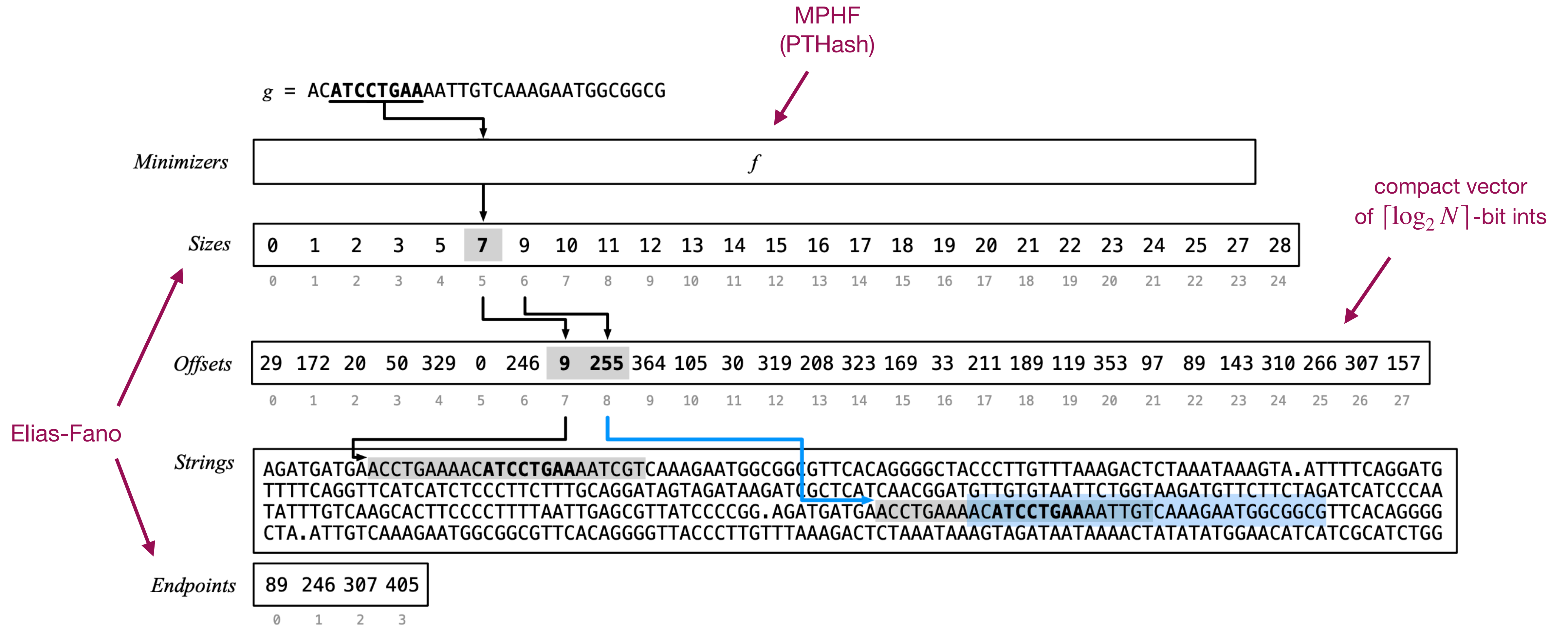
```
TCGTCAA: 29
CATCCAA: 172
ATCGTCAA: 20
GACTCTAA: 50 329
AACCTGAA: 0 246
ATCCTGAA: 9 255
GAACATCA: 364
GCAGGATA: 105
AGGGGCTA: 30
CTTGTTTA: 319
GAGCGTTA: 208
TTTAAAGA: 323
CTTCTAGA: 169
GGCTACCC: 33
CGTTATCC: 211
AGCACTTC: 189
AAGATCGC: 119
AACTATAT: 353
CCTTCTTT: 97
TTCAGGTT: 89
ACGGATGT: 143
ACAGGGGT: 310
TGTCAAAG: 266 307
TAATTCTG: 157
```



offsets



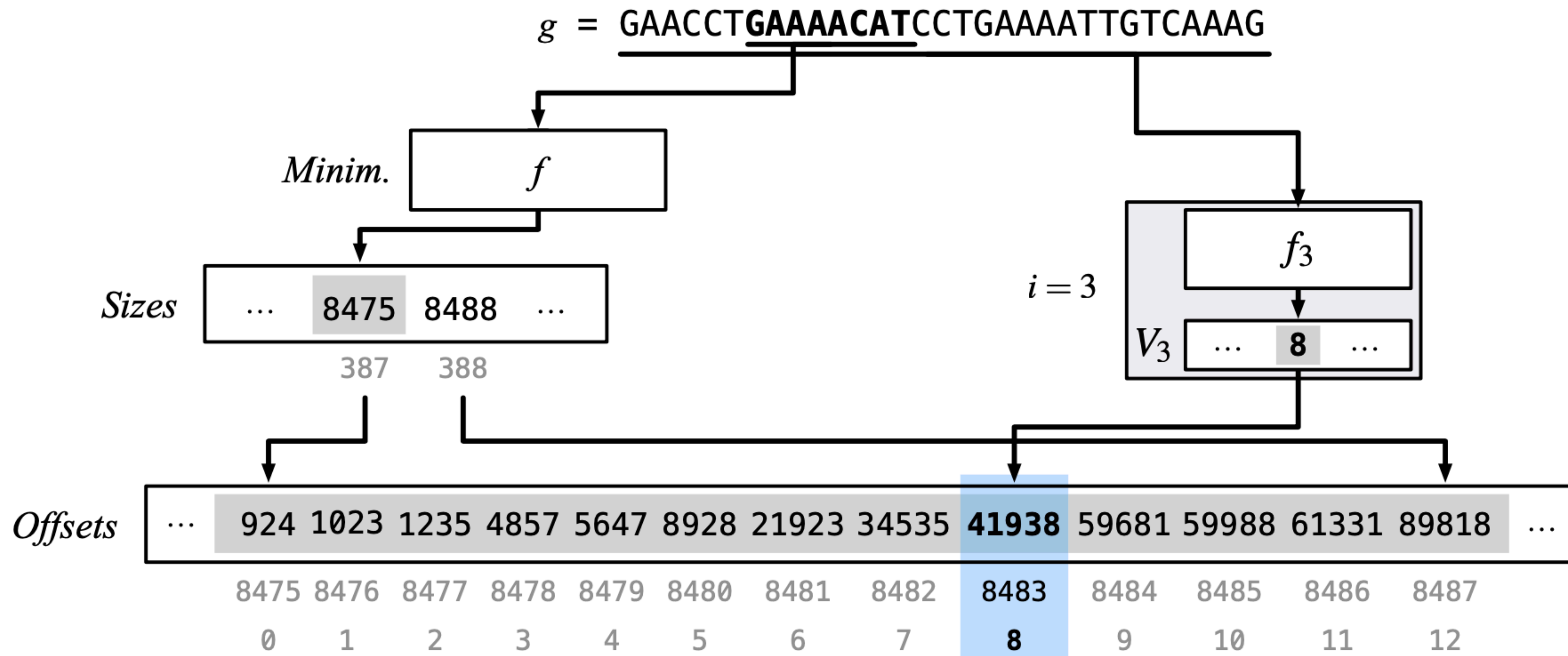
# Sparse Hashing — Example





# Skew Hashing — Example

Example for  $\ell = 3$ .



# Elias-Fano Encoding

- Elias-Fano [Elias, 1974; Fano, 1971] is a succinct data structure representing a monotone integer list  $X[0..n)$  in  $n \lceil \log_2(U/n) \rceil + 2n$  bits, where  $U$  is such that  $U \geq X[n - 1]$ .
- With just  $+o(n)$  extra bits: random Access in  $O(1)$  and Predecessor queries in  $O(\log(U/n))$ .
- Found to be crucial for many practical data structures/applications (e.g., inverted indexes, compressed tries, MPHF).
- See Section 3.4 of [Techniques for Inverted Index Compression](#) P. and Venturini, ACM Computing Surveys, 2021.
- [https://github.com/jermp/data\\_compression\\_course](https://github.com/jermp/data_compression_course)

# Skew Hashing

- For  $i = \ell, \dots, L$ , let  $K_i$  is the set of all  $k$ -mers belonging to buckets of size  $s$ , with  $s$  such that:

$$\begin{cases} 2^i < s \leq 2^{i+1} & \ell \leq i < L \\ 2^L < s \leq \max & i = L \end{cases} .$$

- We build a MPHF  $f_i$  for each set  $K_i$ . For a  $k$ -mer  $g \in K_i$ , we know that its bucket contains at most  $2^{i+1}$  super- $k$ -mers, so we write the identifier of the super- $k$ -mer containing  $g$  in a (compact) vector  $V_i$  of  $(i + 1)$ -bit ints.
- Upon Lookup, we will scan **one** super- $k$ -mer only.

# Trade-offs by Varying Minimizer Length

Space in bits/ $k$ -mer (bpk) and Lookup time (indicated by  $Lkp^+$  for positive queries; by  $Lkp^-$  for negative) in average ns/ $k$ -mer for regular and canonical SShash dictionaries by varying minimizer length  $m$ . For each dataset, we indicate promising configurations in bold font.

Dataset	$m$			$m$			$m$			$m$		
	bpk	$Lkp^+$	$Lkp^-$	bpk	$Lkp^+$	$Lkp^-$	bpk	$Lkp^+$	$Lkp^-$	bpk	$Lkp^+$	$Lkp^-$
Cod	15			<b>16</b>			<b>17</b>			18		
regular	6.60	1236	1267	6.82	1100	1174	<b>6.98</b>	<b>1045</b>	<b>1158</b>	7.21	1015	1157
canonical	7.68	945	768	<b>7.92</b>	<b>834</b>	<b>690</b>	8.18	786	672	8.47	755	658
Kestrel	<b>16</b>			<b>17</b>			18			19		
regular	6.19	1137	1323	<b>6.48</b>	<b>1042</b>	<b>1265</b>	6.79	1005	1245	7.12	997	1240
canonical	<b>7.30</b>	<b>882</b>	<b>781</b>	7.68	790	722	8.09	743	696	8.51	730	691
Human	17			18			<b>19</b>			<b>20</b>		
regular	7.44	1591	1668	7.67	1459	1573	7.95	1406	1547	<b>8.28</b>	<b>1338</b>	<b>1530</b>
canonical	8.76	1150	936	9.04	1054	881	<b>9.39</b>	<b>990</b>	<b>854</b>	9.80	958	838
Bacterial	18			<b>19</b>			<b>20</b>			21		
regular	7.42	1535	1867	7.80	1425	1813	<b>8.22</b>	<b>1389</b>	<b>1780</b>	8.70	1368	1774
canonical	8.75	1129	1043	<b>9.22</b>	<b>1051</b>	<b>995</b>	9.75	1028	947	10.34	998	956

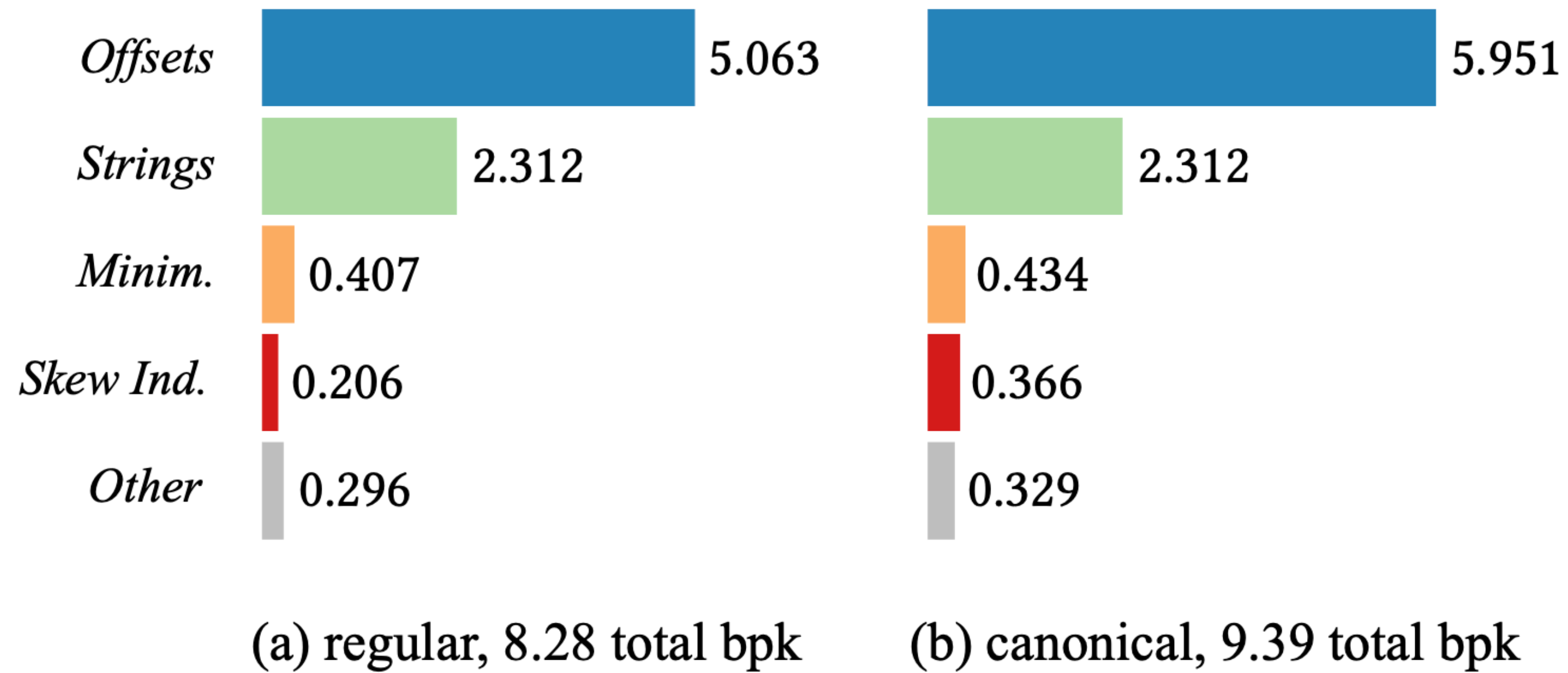
## NOTE 1:

We used  $\ell = 6$  and  $L = 12$  for all experiments.

## NOTE 2:

A good rule of thumb is  
 $m = \lceil \log_4(N) \rceil + 1$  or  
 $m = \lceil \log_4(N) \rceil + 2$ .

# Space Breakdowns



Space breakdowns for the Human dataset, for both (a) regular and (b) canonical dictionaries. The numbers next to each bar indicate the bits/*k*-mer (bpk) spent by the respective components.



# Construction Time and Space

Dictionary construction times in minutes (using a single processing thread) and peak internal memory used during construction in GB. (Blight's performance was the same for all values of  $b$  in the experiment.)

Dictionary	Cod		Kestrel		Human		Bacterial	
	min	GB	min	GB	min	GB	min	GB
dBG-FM, $s = 128$	28.5	0.5	100.0	0.7	–	–	–	–
dBG-FM, $s = 64$	28.5	0.6	100.0	0.9	–	–	–	–
dBG-FM, $s = 32$	28.5	0.7	100.0	1.1	–	–	–	–
Pufferfish, sparse	15.5	3.3	35.2	6.7	86.0	19.4	200.8	40.1
Pufferfish, dense	13.0	2.8	29.2	5.9	70.7	14.0	173.2	30.4
Blight	5.0	3.3	11.0	7.0	25.0	7.5	50.0	15.8
SSHash, regular	1.5	2.6	3.8	5.7	12.5	15.4	29.6	33.4
SSHash, canonical	2.0	2.8	4.4	5.8	16.2	17.3	36.0	36.6

**NOTE:** SShash construction works entirely in internal memory. (This is going to change in future releases.)