On Weighted K-Mer Dictionaries

Giulio Ermanno Pibiri

ISTI-CNR, giulio.ermanno.pibiri@isti.cnr.it

@giulio_pibiri

@jermp

Tutorials on Data Structures for Text Indexation and Compression (TUDASTIC)
Lille, France, 9-10 May 2022

Agenda

1. Context, Motivations, and Problems

Massive DNA Collections

- **Peta bytes** of data available:
  - ENA (European Nucleotide Archive)
  - SRA (Sequence Read Archive)
  - RefSeq (Reference Sequence Database)
  - Ensembl

- For example: as of Feb. 2022, ENA has 2.7 billions of assembled sequences, for >12.6 trillion bases.
  https://www.ebi.ac.uk/ena/browser/about/statistics

- These collections are paving the way to answer fundamental questions regarding biology and evolution.

K-Mers

- **Q.** But how do we exploit such potential?
  We need efficient methods to index and search data at this scale.
- One popular strategy: “reduce” a DNA sequence to a set of short sub-strings of fixed length k — the so-called k-mers.

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

← Example for $k = 13$.

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
  - …

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.

The Weighted K-Mer Dictionary Problem

- We are given a large string over the alphabet {A,C,G,T} (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 $k = 31$.

- K is a set of key-value pairs $\langle g, w(g) \rangle$, where g is a k-mer and $w(g)$ is the number of occurrences — the *weight* — of g in the input.

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

  (Other operations of interest are *iteration* and *streaming* membership queries.)

The Weighted K-Mer Dictionary Problem

- We are given a large string over the alphabet {A,C,G,T} (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 $k = 31$.

- K is a set of key-value pairs $\langle g, w(g) \rangle$, where g is a k-mer and $w(g)$ is the number of occurrences — the *weight* — of g in the input.

- **Problem.** We want to build a dictionary for K so that the following operations are efficient:
  - $i =$ Lookup(g), where $0 \leq i < n$ if $g \in K$ or $i = -1$ otherwise; ← Part **2.**
  - $g =$ Access(i) if $0 \leq i < n$;
  - $w(g) =$ Count(g) if $g \in K$. ← Part **3.**

  (Other operations of interest are *iteration* and *streaming* membership queries.)

2. Sparse and Skew Hashing of K-Mers

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.

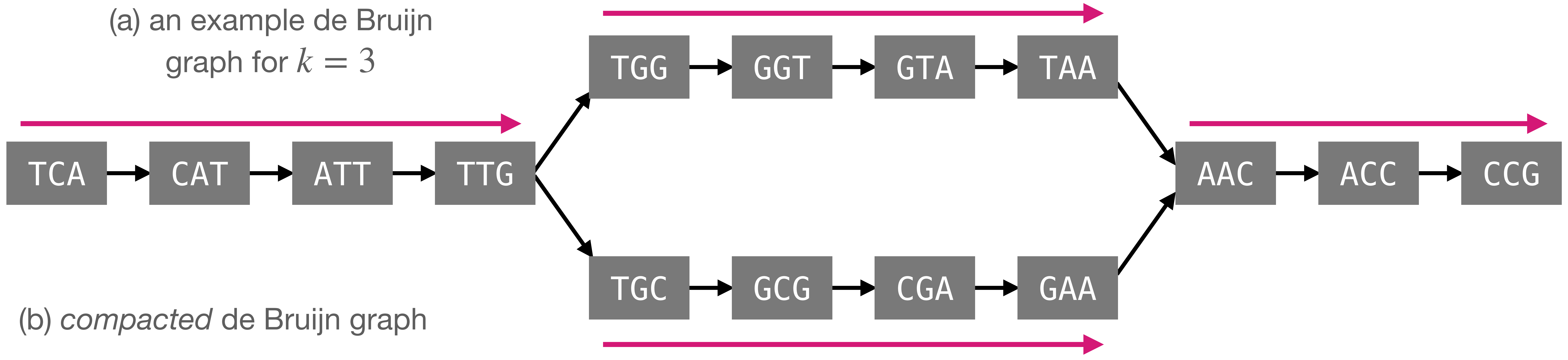
- Since k-mers are extracted *consecutively* from DNA, a k-mer following another one shares $k-1$ bases (very low entropy).

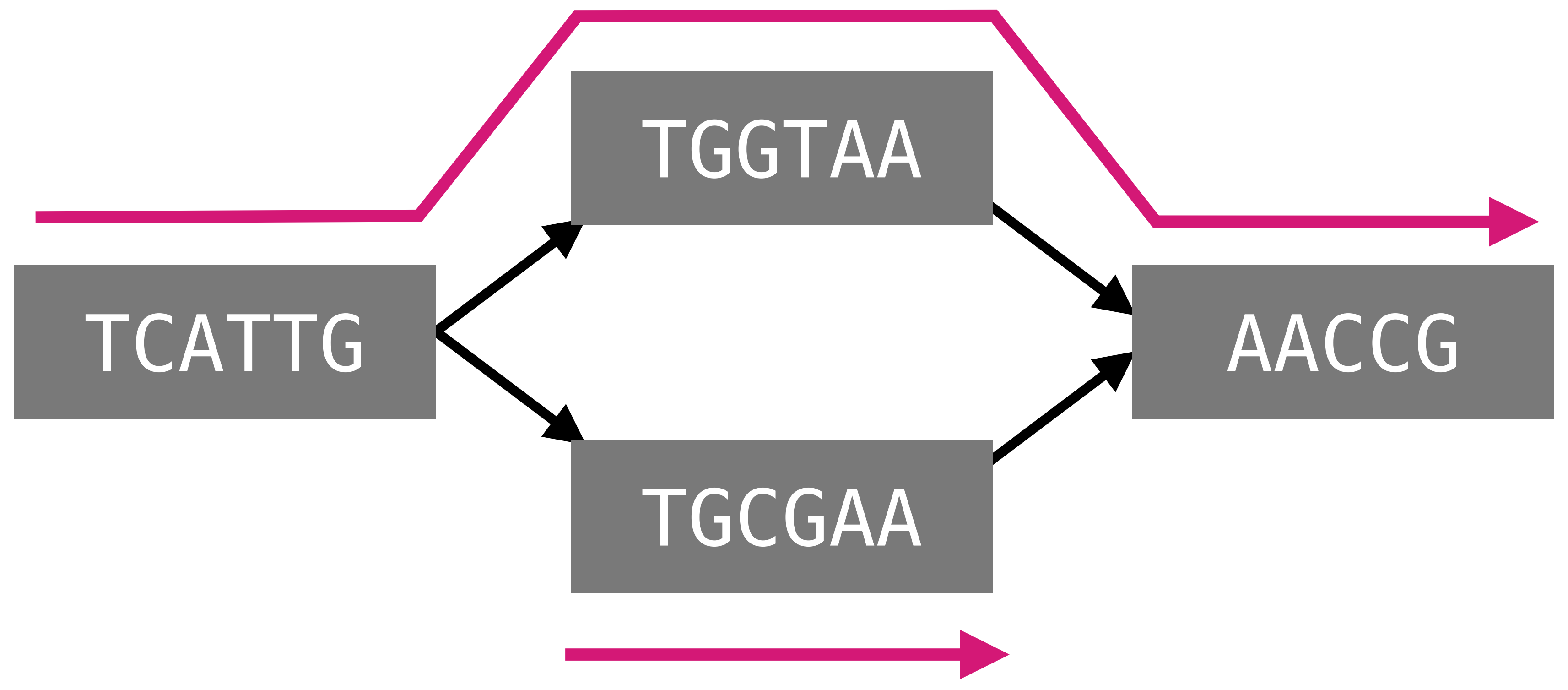
```
ACGGTAGAACCGATTCAAATTCGACGTAGC...
ACGGTAGAACCGA
 CGGTAGAACCGAT
  GGTAGAACCGATT
   GTAGAACCGATTC
    TAGAACCGATTCA
     ...
```

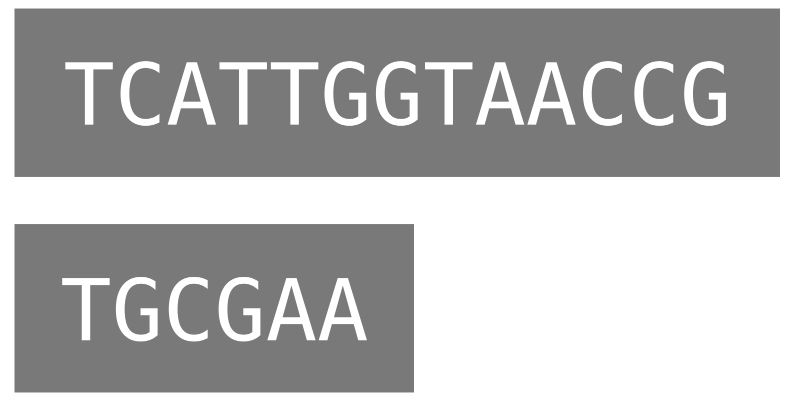
$\longleftarrow$ Example for $k = 13$.

de Bruijn Graphs

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

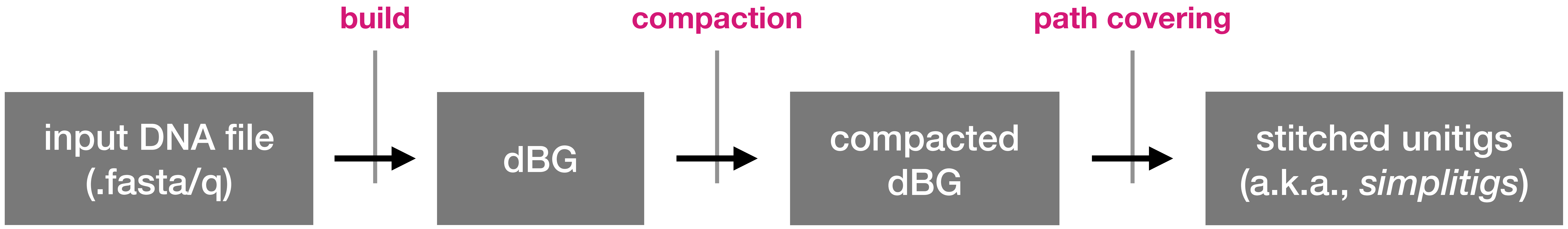






de Bruijn Graphs

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


build
compaction
path covering
input DNA file (.fasta/q)
dBG
compacted dBG
stitched unitigs (a.k.a., *simplitigs*)

- 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].

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 $g =$ `ACGGTAGAACCGA` $(k = 13)$ and $m = 4$:

```
ACGG
 CGGT
  GGTA
   GTAG
    TAGA
     AGAA
      GAAC
       AACC
        ACCG
         CCGA
```

If R is the *lexicographic* order.

```
h(ACGG) = 9842978325
 h(CGGT) = 817612312
  h(GGTA) = 8265731  ←——  smallest hash code
   h(GTAG) = 478491248
    h(TAGA) = 17491411
     h(AGAA) = 17148914
      h(GAAC) = 91815379
       h(AACC) = 645793914
        h(ACCG) = 918417644
         h(CCGA) = 814188124
```

If R is defined by a random hash function h.

Super-k-Mers

- **Property.** Consecutive k-mers are likely to have the same minimizer.

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

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

  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.

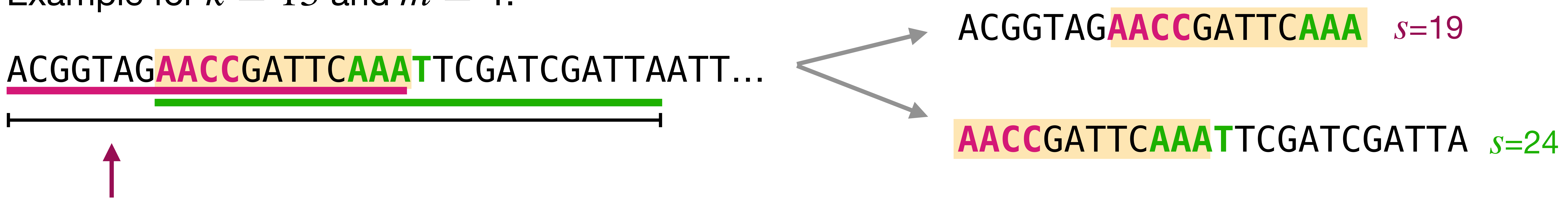
Super-k-Mers

- **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 — $\rightarrow$ **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.

This super-k-mer costs 2x19=38 bits for 7 k-mers (**5.43** bits/k-mer vs. 2x13=26 bits/k-mer).

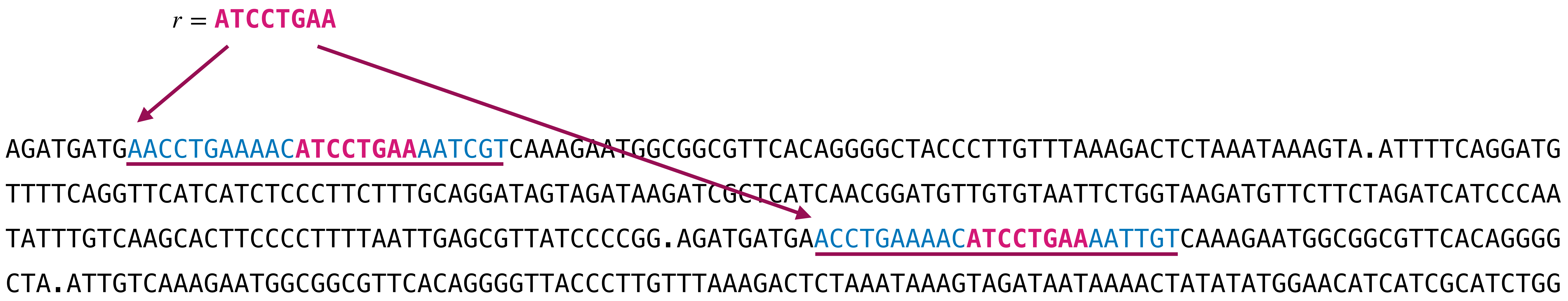
Example for $k = 13$ and $m = 4$:



This **chain** is of length 31 and costs 2x31=62 bits for 19 k-mers (**3.26** bits/k-mer).

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. (An offset is an integer in $[0, N)$, where N is the number of bases in the strings.)
- Upon 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.



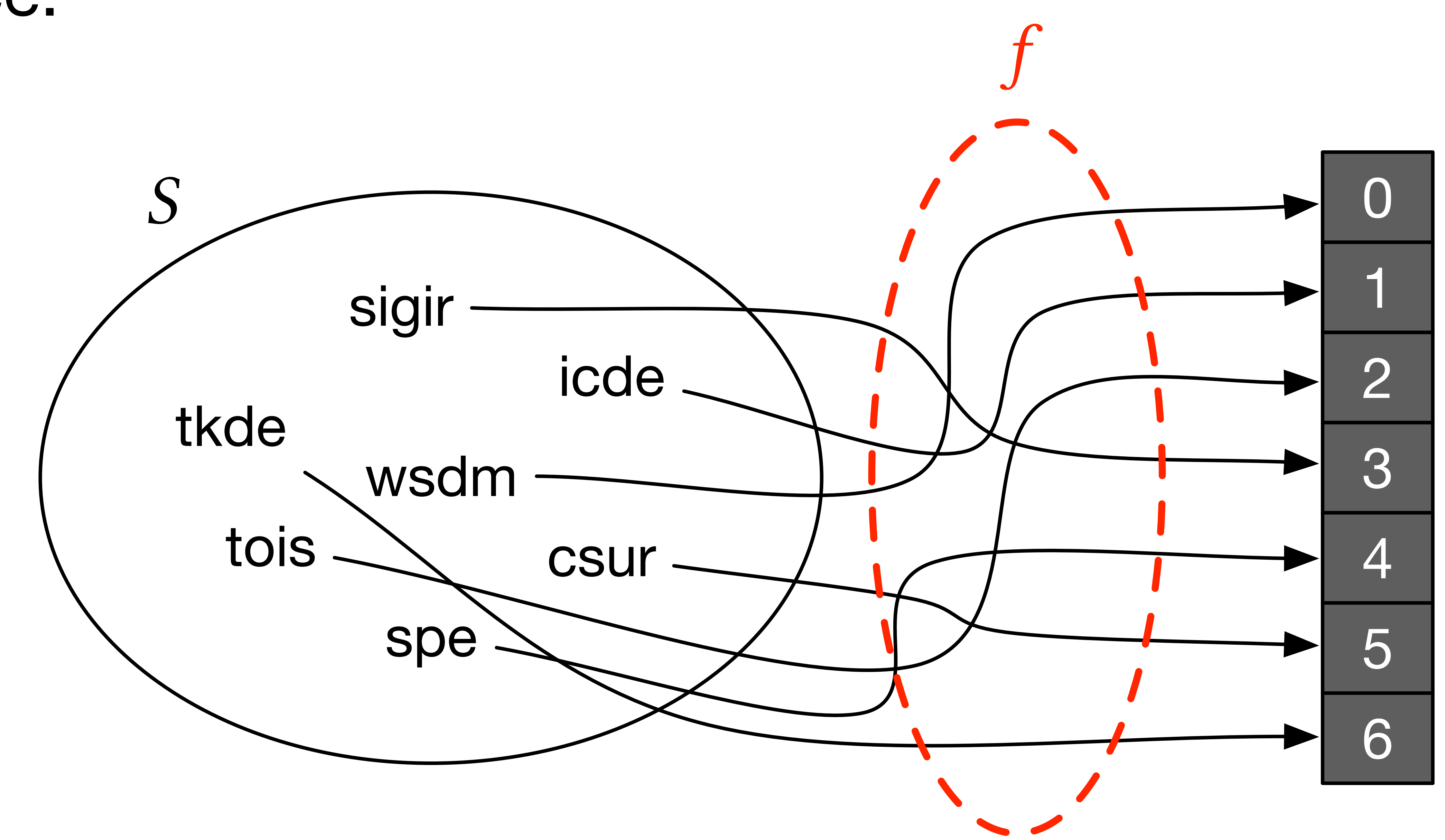
```
AGATGATGAACCTGAAAACATCCTGAAAATCGTCAAAGAATGGCGGCGTTCACAGGGGCTACCCTTGTTTAAAGACTCTAAATAAAGTA.ATTTTCAGGATG
TTTTCAGGTTCATCATCTCCCTTCTTTGCAGGATAGTAGATAAGATCGCTCATCAACGGATGTTGTGTAATTCTGGTAAGATGTTCTTCTAGATCATCCCAA
TATTTGTCAAGCACTTCCCCTTTTAATTGAGCGTTATCCCCGG.AGATGATGAACCTGAAAACATCCTGAAAATTGTCAAAGAATGGCGGCGTTCACAGGGG
CTA.ATTGTCAAAGAATGGCGGCGTTCACAGGGGTTACCCTTGTTTAAAGACTCTAAATAAAGTAGATAATAAAACTATATATGGAACATCATCGCATCTGG
```

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 $\{0,\ldots,n-1\}$ 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

24 minimizers, for $m = 8$

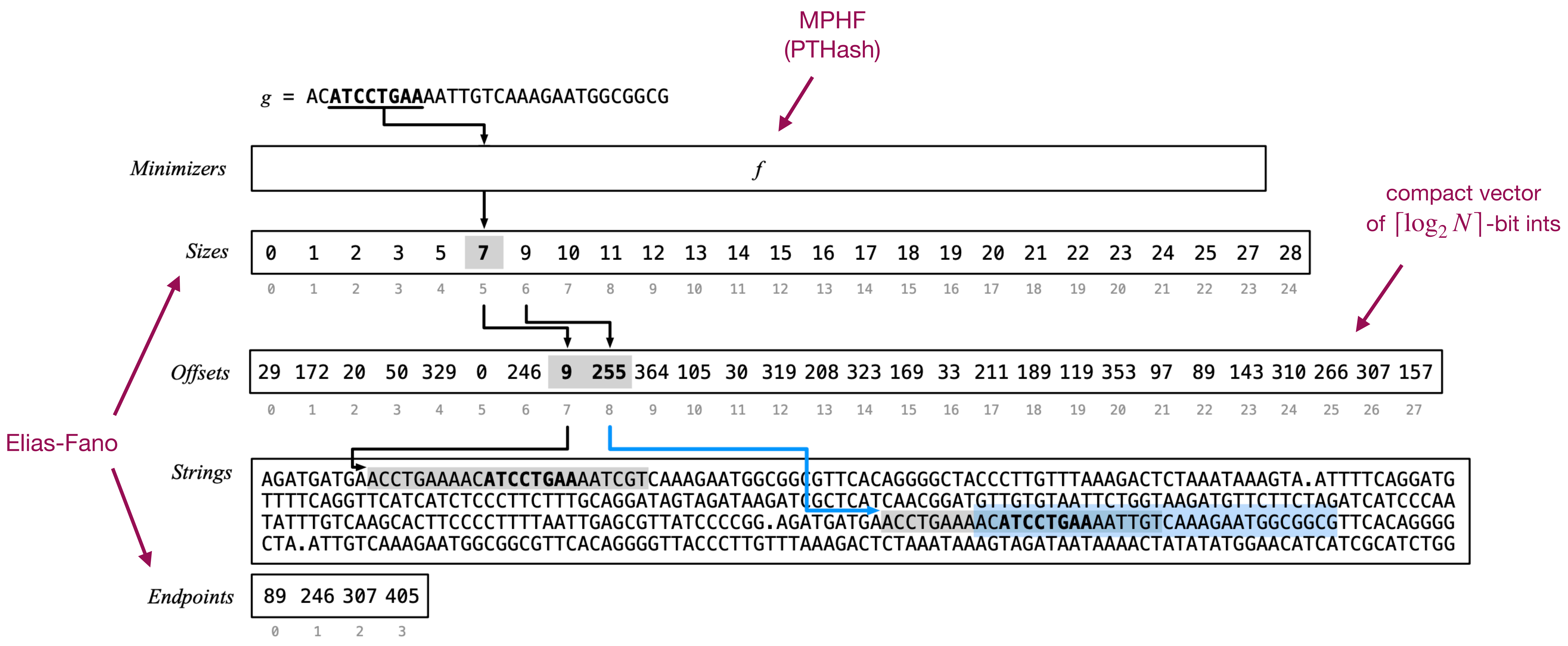
a collection of 4 stitched unitigs:
285 k-mers for $k = 31$, $N = 408$ bases

```
AGATGATGAACCTGAAAACATCCTGAAAATCGTCAAAGAATGGCGG
CGTTCACAGGGGCTACCCTTGTTTAAAGACTCTAAATAAAGTA.AT
TTTCAGGATGTTTTCAGGTTCATCATCTCCCTTCTTTGCAGGATAG
TAGATAAGATCGCTCATCAACGGATGTTGTGTAATTCTGGTAAGAT
GTTCTTCTAGATCATCCCAATATTTGTCAAGCACTTCCCCTTTTAA
TTGAGCGTTATCCCCGG.AGATGATGAACCTGAAAACATCCTGAAA
ATTGTCAAAGAATGGCGGCGTTCACAGGGGCTA.ATTGTCAAAGAA
TGGCGGCGTTCACAGGGGTTACCCTTGTTTAAAGACTCTAAATAAA
GTAGATAATAAAACTATATATGGAACATCATCGCATCTGG
```

```
TCGTCAAA: 29
CATCCCAA: 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



g = AC**ATCCTGAA**AATTGTCAAAGAATGGCGGCG

Sizes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	1	2	3	5	**7**	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	27	28

Offsets

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
29	172	20	50	329	0	246	**9**	**255**	364	105	30	319	208	323	169	33	211	189	119	353	97	89	143	310	266	307	157

Strings

```
AGATGATGAACCTGAAAACATCCTGAAAATCGTCAAAGAATGGCGGCGTTCACAGGGGCTACCCTTGTTTAAAGACTCTAAATAAAGTA.ATTTTCAGGATG
TTTTCAGGTTCATCATCTCCCTTCTTTGCAGGATAGTAGATAAGATCGCTCATCAACGGATGTTGTGTAATTCTGGTAAGATGTTCTTCTAGATCATCCCAA
TATTTGTCAAGCACTTCCCCTTTTAATTGAGCGTTATCCCCGG.AGATGATGAACCTGAAAACATCCTGAAAATTGTCAAAGAATGGCGGCGTTCACAGGGG
CTA.ATTGTCAAAGAATGGCGGCGTTCACAGGGGTTACCCTTGTTTAAAGACTCTAAATAAAGTAGATAATAAAACTATATATGGAACATCATCGCATCTGG
```

Endpoints

0	1	2	3
89	246	307	405

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

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 3.6×10^4.

- **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 ℓ: by virtue of the skew distribution, the fraction of buckets having **more than** 2^{ℓ} super-k-mers is **small**.
- So, we can afford a MPHF over the set of k-mers that belong to such super-k-mers.

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.

Skew Hashing

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

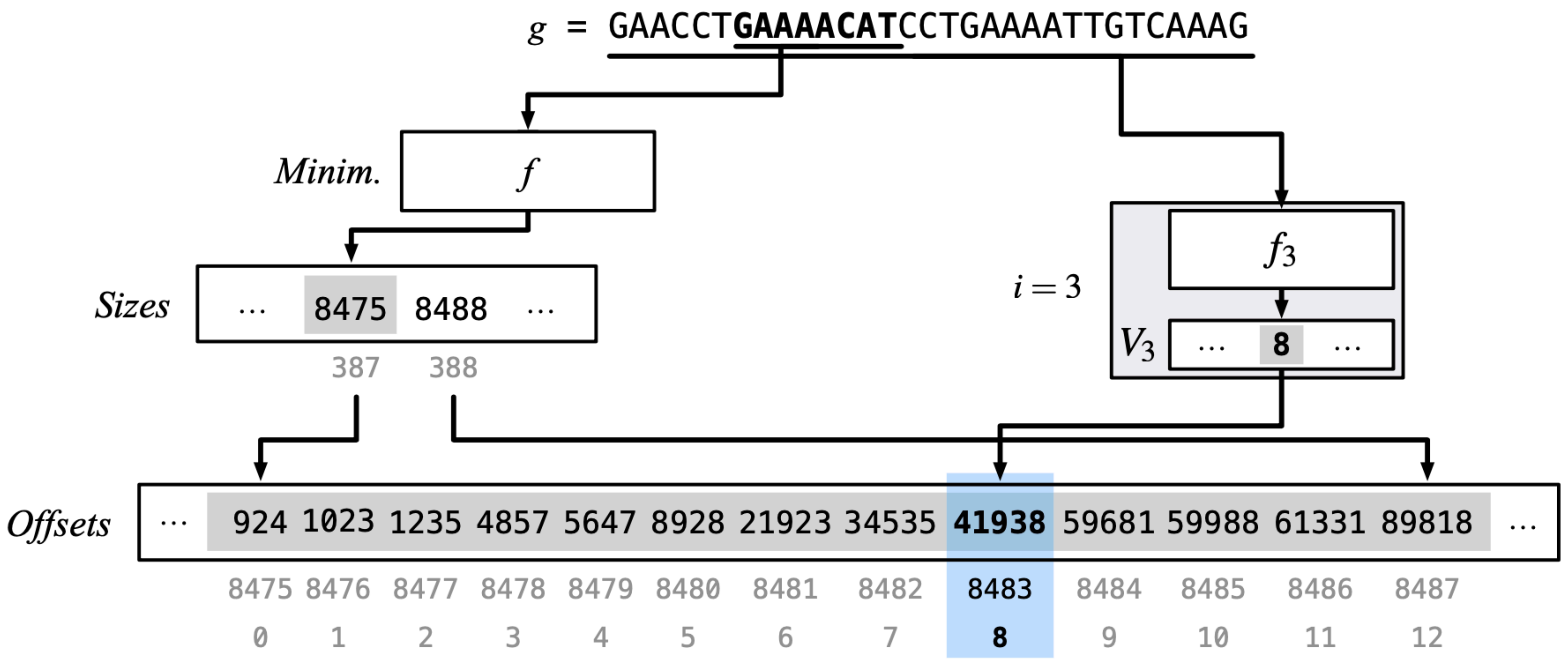
$$\begin{cases} 2^i < s \leq 2^{i+1} & \ell \leq i < L \\ 2^L < s \leq max & i = L \end{cases}.$$

- 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.

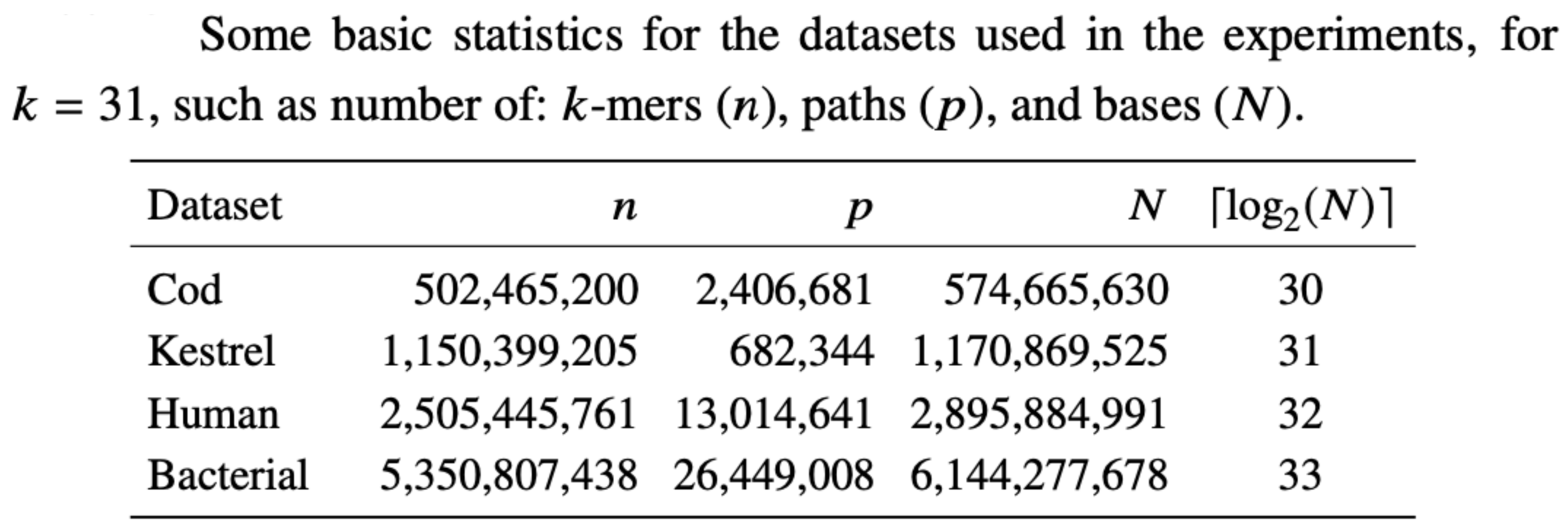
Skew Hashing — Example

Example for $\ell = 3$.



Experimental Setup and Datasets

- Processor: Intel(R) Core(TM) i9-9940X CPU @ 3.30GHz
- Compiler and OS: gcc version 11.2.0, Ubuntu 11.2.0-7ubuntu2
- Code in C++17, compiled with flags: `-O3 -march=native`

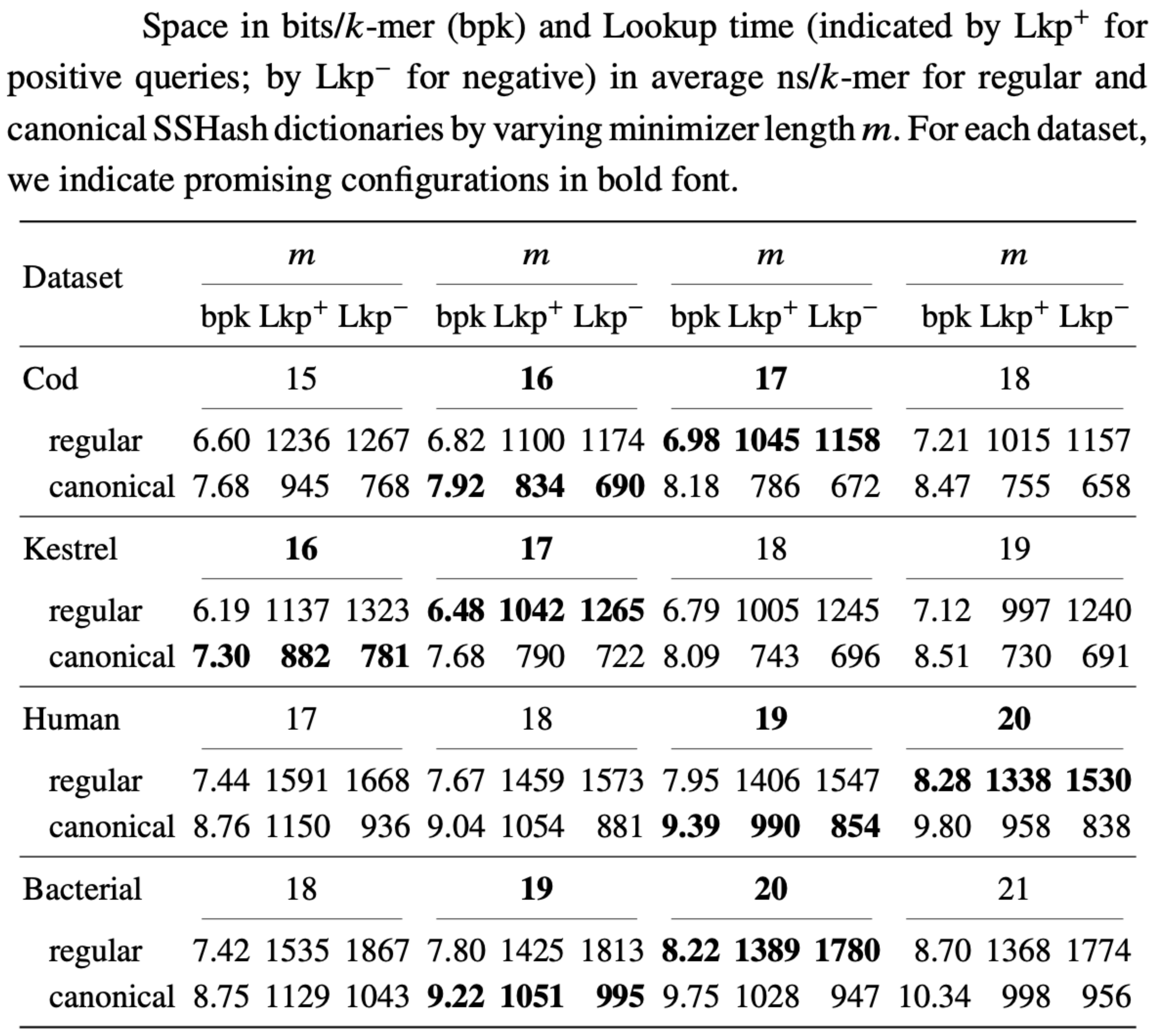


Some basic statistics for the datasets used in the experiments, for $k = 31$, such as number of: k-mers (n), paths (p), and bases (N).

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.

Trade-offs by Varying Minimizer Length



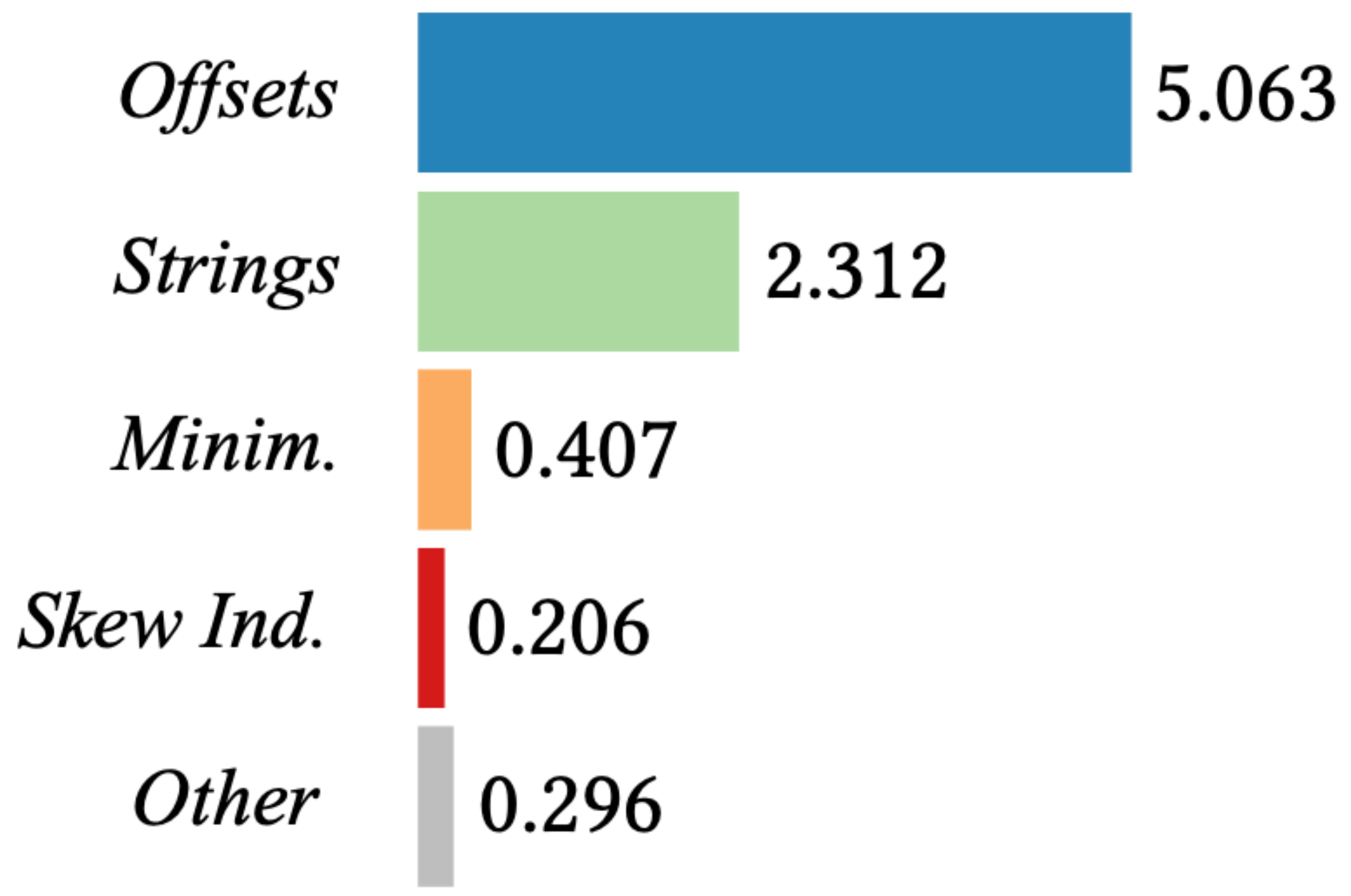
Space in bits/k-mer (bpk) and Lookup time (indicated by Lkp$^+$ for positive queries; by Lkp$^-$ for negative) in average ns/k-mer for regular and canonical SSHash dictionaries by varying minimizer length m. For each dataset, we indicate promising configurations in bold font.

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

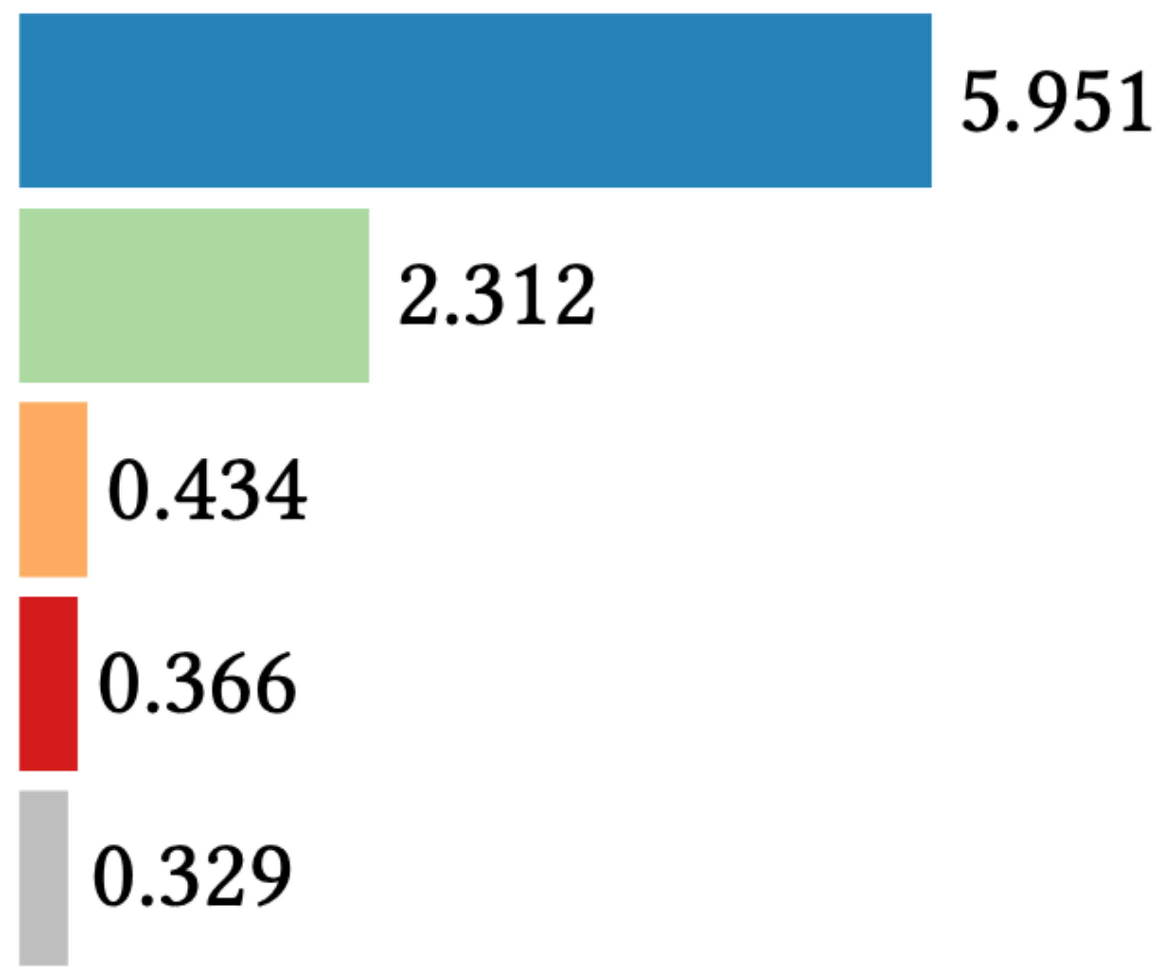
NOTE 1:
We used $\ell = 6$ and $L = 12$ for all experiments.

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



(a) regular, 8.28 total bpk



(b) canonical, 9.39 total bpk

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.

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$^+$	Lkp$^-$	Lkp$^+$	Lkp$^-$	Lkp$^+$	Lkp$^-$	Lkp$^+$	Lkp$^-$
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	7929	11,703	8073	–	–	–	–
Pufferfish, sparse	1110	700	5456	769	13,656	862	27,748	983
Pufferfish, dense	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

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 81.37% hits		SRR11449743 74.60% hits		SRR5833294 91.65% hits		SRR5901135 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 0.659% hits		SRR12858649 0.484% hits		SRR5901135 0.002% hits		SRR5833294 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

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.)

3. Weight Compression

SSHash is Order-Preserving

- **Quick Recap.** For a set K of n distinct k-mers, SSHash implements a function (Lookup) $h : \Sigma^k \to \{-1, 0, \ldots, n-1\}$, where $0 \le h(g) < n$ if $g \in K$ and $h(g) = -1$ if $g \notin K$.

- **Order-Preserving Property.** If g_2 if the successor of g_1, then $h(g_2) = h(g_1) + 1$.

- This is a direct consequence of indexing a *spectrum-preserving string set* (SPSS): K is reduced to a set of p strings $\mathcal{S} = \{S_0, \ldots, S_{p-1}\}$.

- Any order on $\mathcal{S}$ uniquely determines an order $i = 0, \ldots, n-1$ for the k-mers $\{g_i\}_i$, thus: $h(g_i) = i$.

The Weights

- Let $W[0..n-1]$ be the sequence of k-mer weights, where $W[i] = w(g_i)$ and $i = h(g_i)$.
- **Property.** The order-preserving property of h makes W have **runs** of equal weights, because consecutive k-mers are likely to have the same weight.
- We exploit the order of the k-mers to preserve the natural order of the weights.

```
>5 5 5 5 5 5 5 5 5 5 5 5 5 5
GGTAATGCAGCCAGGGATGCAACGACCGCAACAGAAAAAGCCCG

>4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 4 4 4 4
CAGCTCATTACAGAAAAAATACCGCTCACCGCCCTGCACCGTCAGGTCAATTTCCCTGAGCACCACCCGCGGTGACTGCTCTGATTTAACC

>4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
CAGCTATGCAGGAGACAAGAATCGCCAGCTTACCCGTTACAGCGATACCCGCTGGCATG

>13 13 13 13 13 13 13
TCAGGTGTACGGTGTGCGTAAAGTCTGGCGTCAGTTG
```

We have 6 runs in this example:
5 (14x), **4** (18x), **2** (8x), **1** (31x), **4** (33x), **13** (7x).

Run-Length Encoding (RLE)

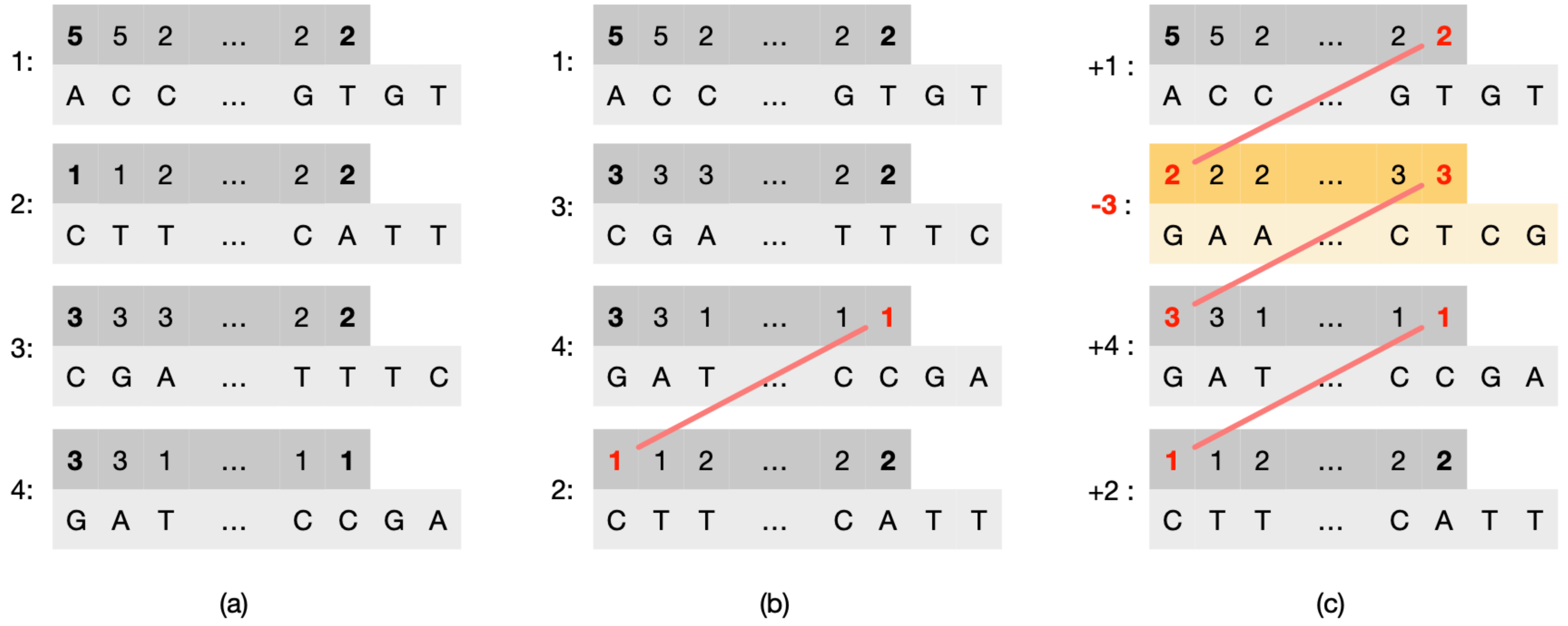
- Represent W with r runs as a sequence of run-length pairs $RLW = \langle w_0, \ell_0 \rangle \langle w_1, \ell_1 \rangle \dots \langle w_{r-1}, \ell_{r-1} \rangle$.
- Take the prefix-sums of the lengths $0, \ell_0, \ell_1, \dots, \ell_{r-2}$ into an array $P[0..r-1]$ and encode it with Elias-Fano.
- We spend, at most

$$r \cdot \left(\underset{\uparrow\ \text{Number of bits dedicated to each } w_i.}{c} + \underbrace{\lceil \log_2(n/r) \rceil + 2 + o(1)}_{\text{Elias-Fano on the lengths.}} \right) \text{ bits for } RLW.$$

- To retrieve $w(g)$ from $i = h(g)$, all that we need is a predecessor query over P which is done in $O(\log(n/r))$ with Elias-Fano.

Reducing the Number of Runs

- **Strategy.** Change the **order** of the strings in $\mathcal{S} = \{S_0, \ldots, S_{p-1}\}$ and possibly take the **reverse-complement** of a string (and reverse the corresponding weights) to reduce the number of runs.
- **Goal.** Compute a signed permutation $\pi[0..p-1]$ where $\pi[i] = j$ indicates that:
  - if $j < 0$: reverse(S_i) has to appear in position $-j$;
  - else: S_i has to appear in position j.

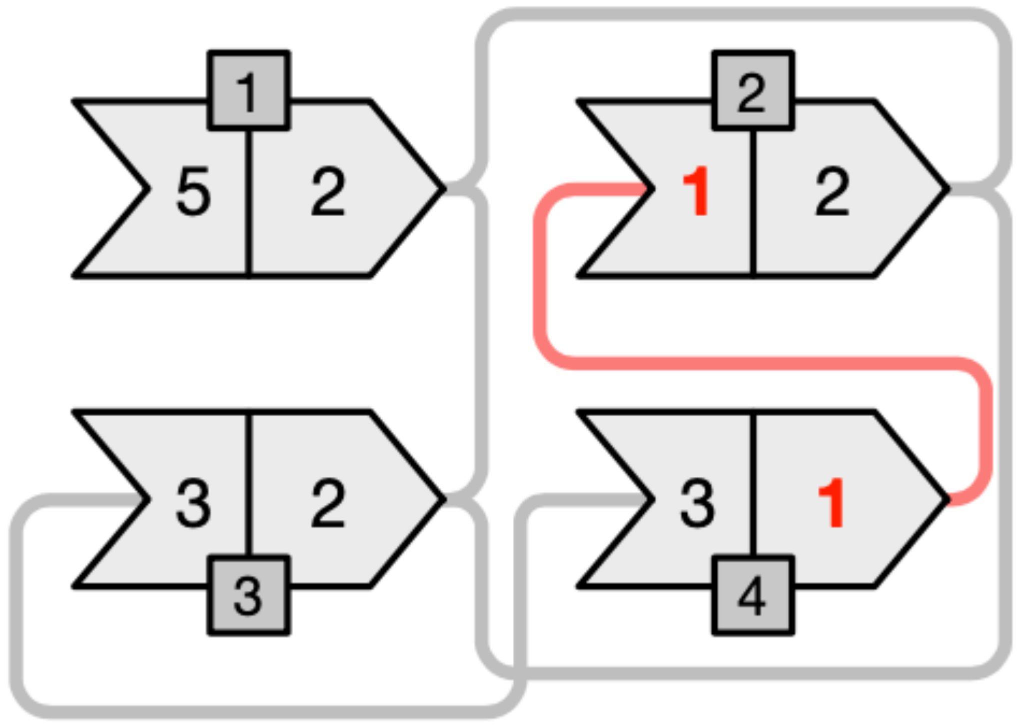


$$\pi = \begin{array}{cccc}[+1, & +4, & -2, & +3] \\ 1 & 2 & 3 & 4\end{array}$$

NOTE: The result π only depends on the **end-point weights** of a string and not on the other weights, nor on the nucleotide sequences.

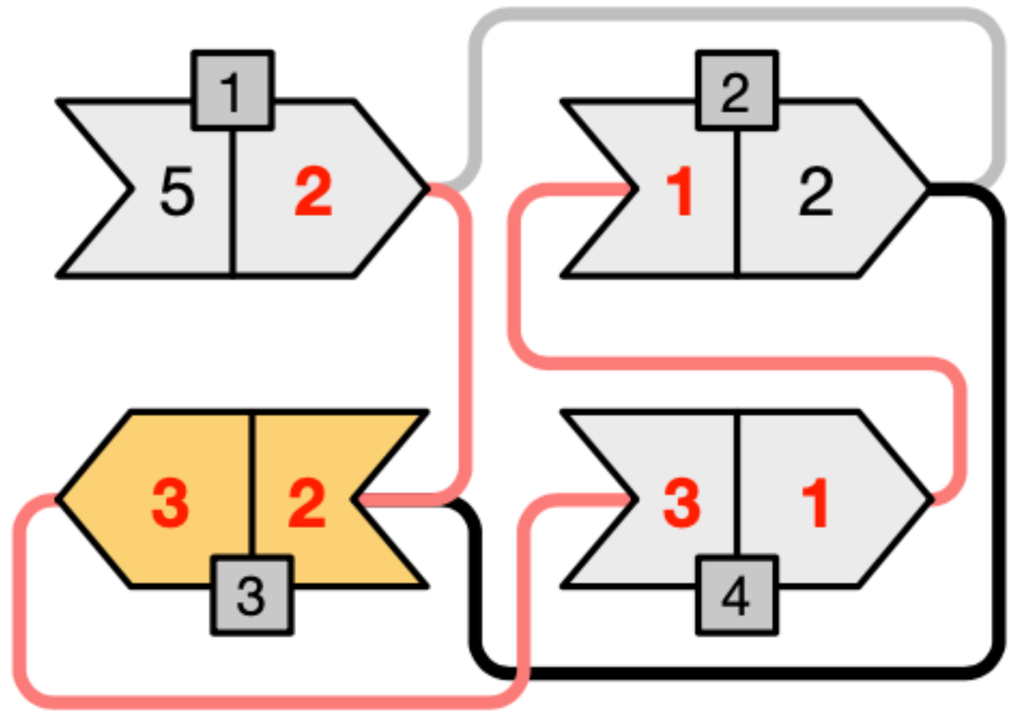
End-Point Weight Graphs and Path Covers

- Since the result π only depends on the end-point weights, it is convenient to consider the **end-point weight graph** $ewG(\mathcal{S})$ for $\mathcal{S}$.
- A (disjoint-node) **path cover** C for $ewG(\mathcal{S})$ determines a signed permutation π.
- Minimizing the number of runs in $\mathcal{S}$ is equivalent to finding a **minimum-cardinality** path cover C for $ewG(\mathcal{S})$.
- We can compute a **lower bound** on the number of runs.



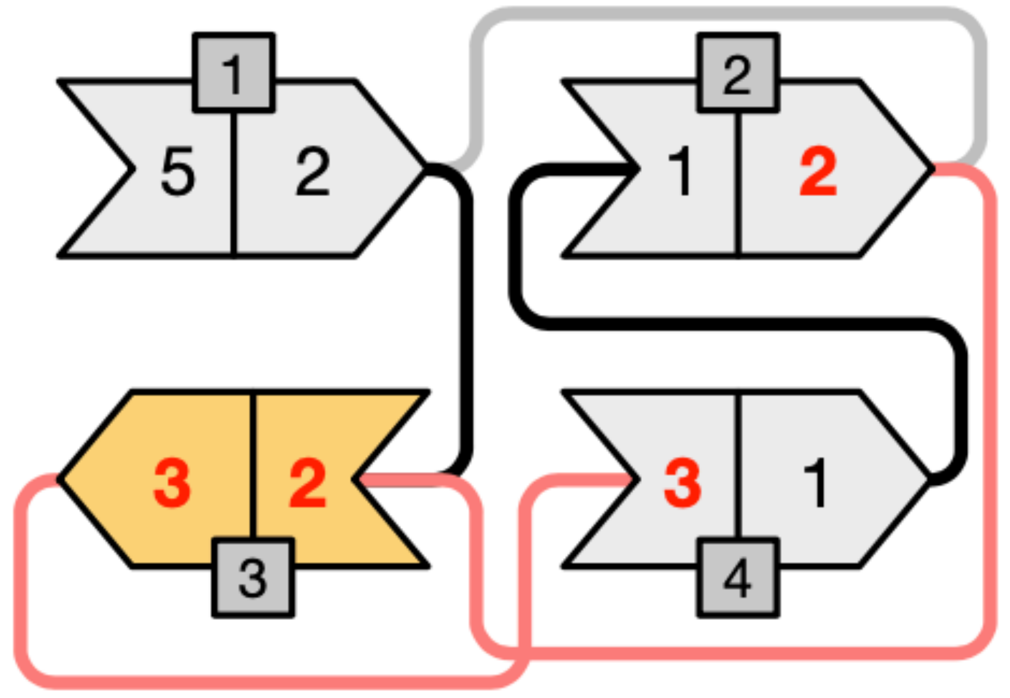
(a)

$\pi = [+1, +4, +2, +3]$



(b)

$\pi = [+1, +4, -2, +3]$



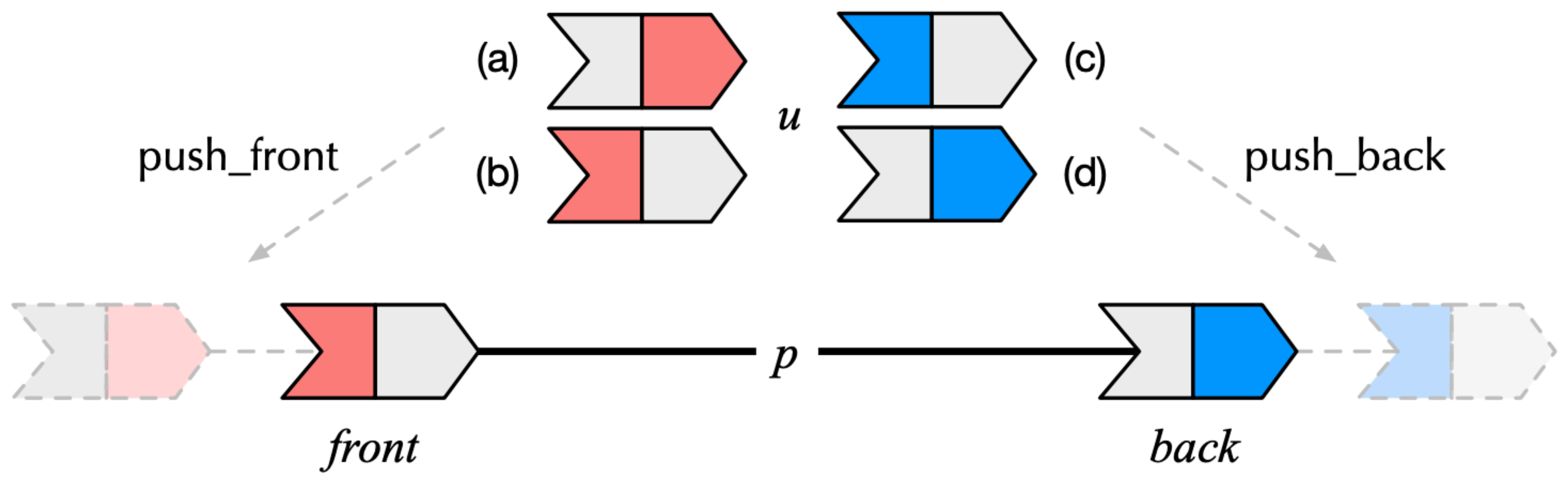
(c)

$\pi = [+4, +1, -2, +3]$

Computing a Path Cover

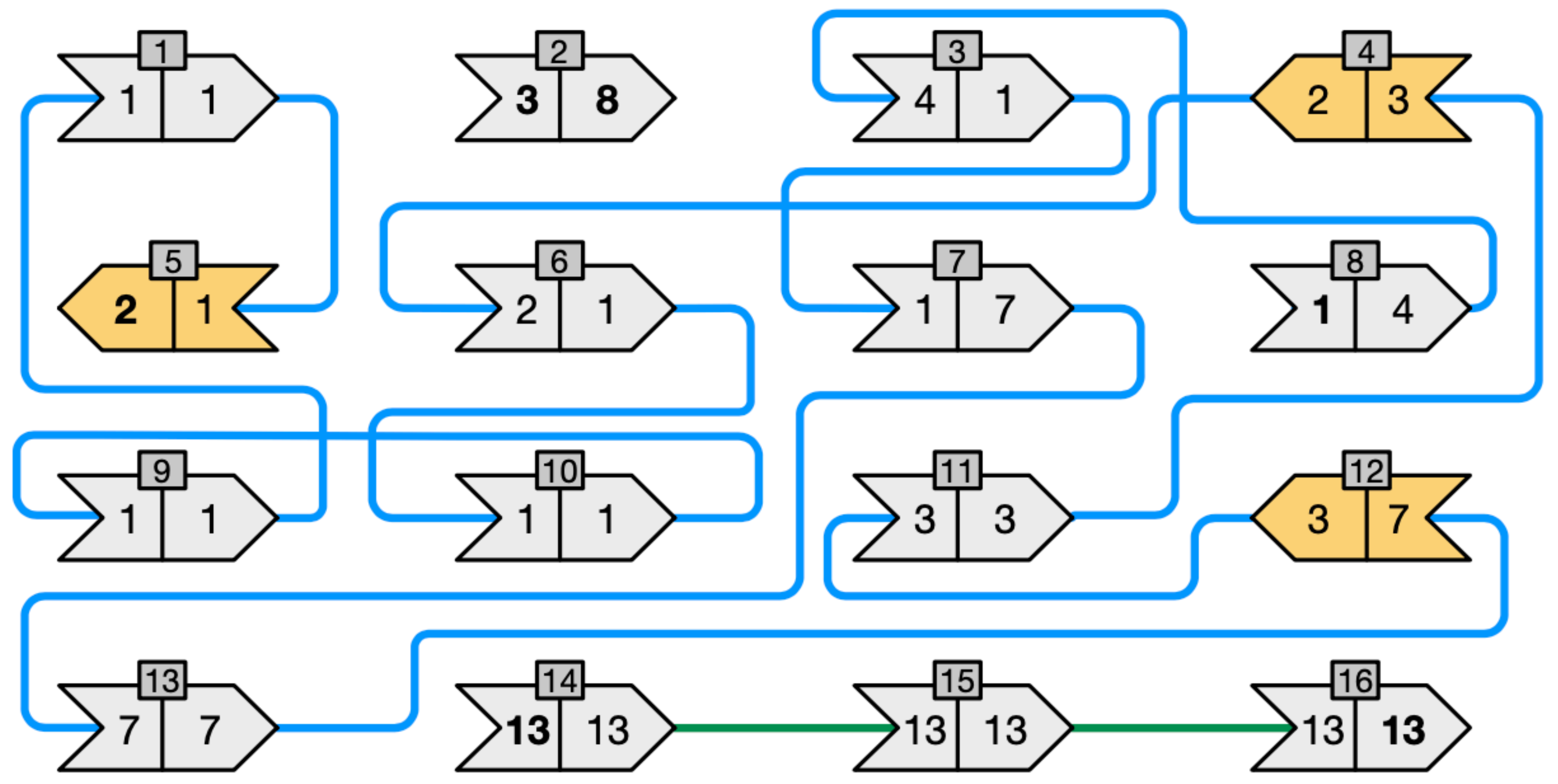
```
cover($ewG(\mathcal{S})$):
    $incidence = \varnothing$
    $unvisited = \varnothing$
    for each node $u \in ewG(\mathcal{S})$:
        $unvisited$.insert($u$)
        $incidence[u.left]$.insert($u$)
        $incidence[u.right]$.insert($u$)
    while $unvisited \neq \varnothing$ :
        $u = unvisited$.take()
        $p = \varnothing$
        while true :
            extend $p$ with $u$
            $unvisited$.erase($u$)
            $incidence[u.left]$.erase($u$)
            $incidence[u.right]$.erase($u$)
            if $incidence[p.back.right] \neq \varnothing$ :
                $u = incidence[p.back.right]$.take()
            else if $incidence[p.front.left] \neq \varnothing$ :
                $u = incidence[p.front.left]$.take()
            else : break
        for each $u \in p$ :
            print $(u.sign, u.id)$
```

- If we use hashing to implement *incidence* and *unvisited*, then insert/erase/take are all supported in $O(1)$ **expected** time.
- So the overall complexity (in both time and space) is **linear** in the number of nodes in $ewG(\mathcal{S})$.



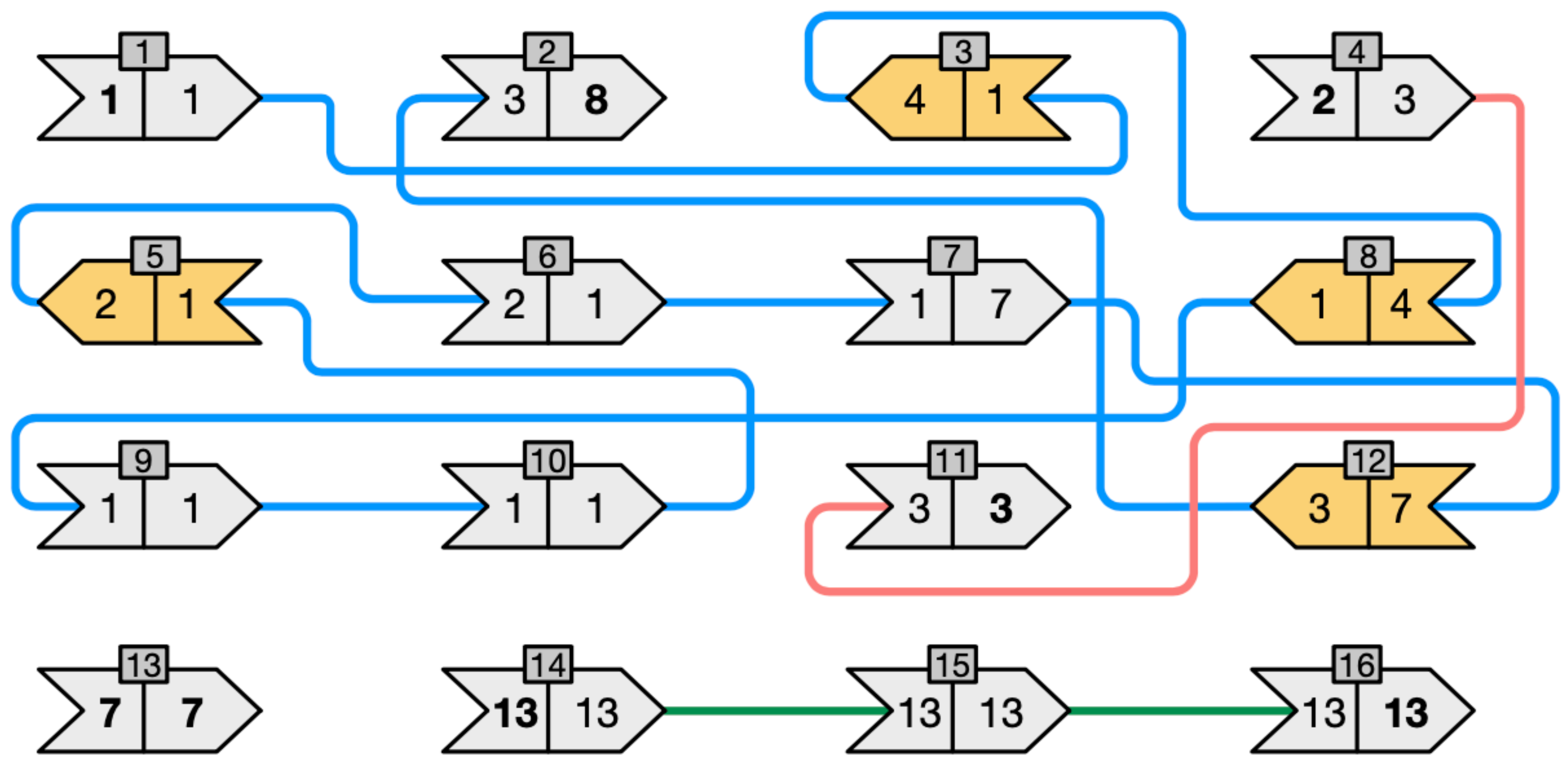
Examples of Path Covers

(a)



C contains **3** paths (**optimal**).

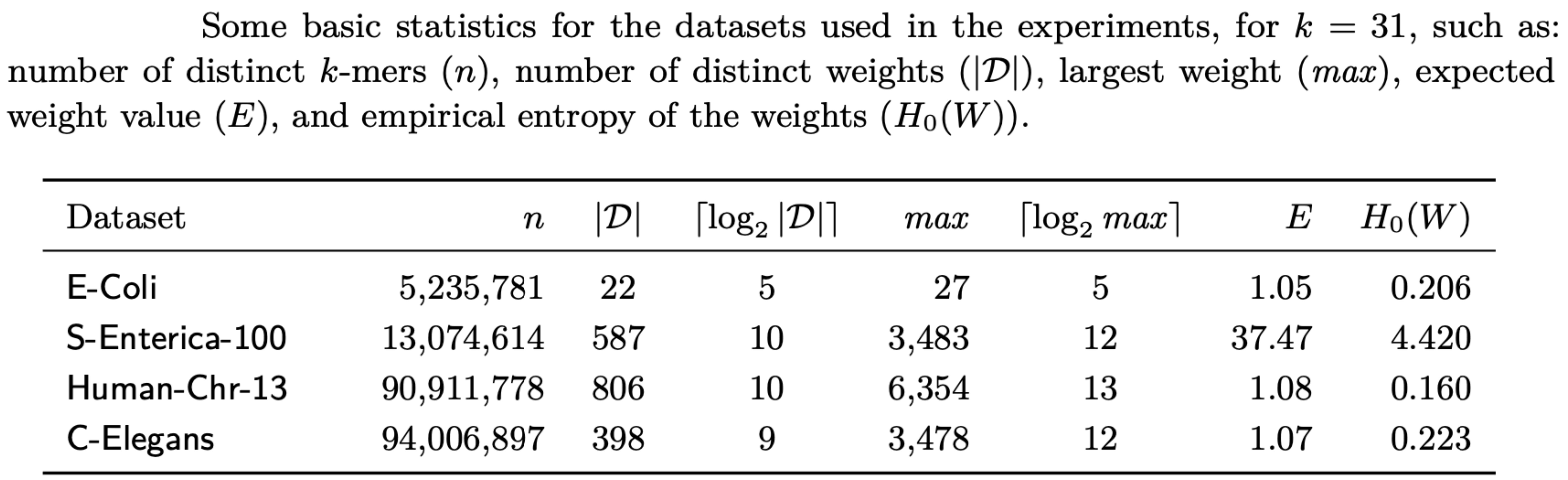
(b)



C contains **4** paths.

Experimental Setup and Datasets

- Processor: Intel(R) Core(TM) i9-9940X CPU @ 3.30GHz
- Compiler and OS: gcc version 11.2.0, Ubuntu 11.2.0-7ubuntu2
- Code in C++17, compiled with flags: `-O3 -march=native`



Some basic statistics for the datasets used in the experiments, for $k = 31$, such as: number of distinct k-mers (n), number of distinct weights ($|\mathcal{D}|$), largest weight (max), expected weight value (E), and empirical entropy of the weights ($H_0(W)$).

Dataset	n	$\lvert\mathcal{D}\rvert$	$\lceil\log_2 \lvert\mathcal{D}\rvert\rceil$	max	$\lceil\log_2 max\rceil$	E	$H_0(W)$
E-Coli	5,235,781	22	5	27	5	1.05	0.206
S-Enterica-100	13,074,614	587	10	3,483	12	37.47	4.420
Human-Chr-13	90,911,778	806	10	6,354	13	1.08	0.160
C-Elegans	94,006,897	398	9	3,478	12	1.07	0.223

Weight Compression

Space for the weights in bits/k-mer, *before* and *after* the run-reduction optimization. In parentheses, we report the compression ratio compared to the empirical entropy.

Dataset	$H_0(W)$	*before*		*after*	
E-Coli	0.206	0.017	$(12.11\times)$	0.014	$(15.10\times)$
S-Enterica-100	4.420	0.592	$(7.47\times)$	0.401	$(11.02\times)$
Human-Chr-13	0.160	0.136	$(1.18\times)$	0.107	$(1.50\times)$
C-Elegans	0.223	0.069	$(3.23\times)$	0.055	$(4.05\times)$

Number of strings (p), number of runs (r) in comparison to the lower bound (r_{lo}), and the run-time of the path cover algorithm (Alg. 3).

Dataset	p	r_{lo}	r	Alg. 3 (ms)	Alg. 3 (ns/node)
E-Coli	2,102	3,723	3,723	0.6	285
S-Enterica-100	150,604	277,649	277,658	53.0	352
Human-Chr-13	266,113	462,175	462,197	94.6	355
C-Elegans	140,452	247,661	247,669	47.1	335

Competitors

- dBG-FM [Chikhi et al., 2014]: FM-index [Ferragina and Manzini, 2000]
- cw-dBG [Italiano et al., 2021]: *weighted* BOSS [Bowe et al., 2012]
- BCFS and AMB [Shibuya et al., 2021]: *compressed static functions* (CSFs) — efficient *maps* from k-mers to weights (the k-mers are not represented)

Overall Comparison

Dictionary space in average bits/k-mer and count time in average μsec/k-mer. For reference, we report in gray color the space and time of SSHash *without* the weight information.

Dictionary	E-Coli		S-Enterica-100		Human-Chr-13		C-Elegans	
	space	query-time	space	query-time	space	query-time	space	query-time
dBG-FM, $s = 128$	3.20	14.73	113.78	16.47	3.23	17.40	3.18	18.05
dBG-FM, $s = 64$	4.02	7.91	142.25	11.13	4.07	11.33	4.01	10.89
dBG-FM, $s = 32$	5.65	4.62	198.71	8.57	5.73	8.20	5.67	7.90
cw-dBG, $s = 128$	2.79	109.13	5.59	120.72	2.80	100.88	2.77	127.86
cw-dBG, $s = 64$	2.86	70.93	5.74	85.73	2.86	73.91	2.84	84.19
cw-dBG, $s = 32$	2.99	52.29	6.03	66.25	2.99	59.85	2.97	62.54
SSHash+BCSF	5.07	0.82	11.12	0.89	6.15	1.25	6.00	1.28
SSHash+AMB	4.90	1.34	9.27	1.65	6.08	1.95	5.88	1.97
w-SSHash	4.80	0.37	6.57	0.48	6.04	0.84	5.75	0.85
SSHash	4.79	0.34	6.15	0.41	5.93	0.76	5.69	0.77

Additional Results for w-SSHash

Number of k-mers, number of strings (p), number of runs (r) in comparison to the lower bound (r_{lo}), and the run-time of the path cover algorithm in total seconds (Alg. 3), index space in bits/k-mers, and query time in μsec/k-mer.

Dataset	n	p	r_{lo}	r	Alg. 3	$H_0(W)$	space	query-time
Cod	502,465,200	2,406,681	4,183,202	4,183,230	1.2	0.441	6.98+0.19	1.3
Kestrel	1,150,399,205	682,344	1,140,743	1,140,747	0.3	0.089	6.49+0.02	1.1
Human	2,505,445,761	13,014,641	22,680,047	22,680,099	7.5	0.453	8.28+0.22	1.6
Bacterial	5,350,807,438	26,448,286	56,662,230	56,662,304	17.2	1.890	8.22+0.24	1.9

4. Conclusions and Future Directions

Conclusions

- SSHash is an efficient solution to the Weighted K-Mer Dictionary problem:
  good trade-off between space and time.

- Tool-box: SPSS, minimizers, MPHF (https://github.com/jermp/pthash), Elias-Fano, RLE.

- Ingredients:
  - Sparse indexing to obtain good space effectiveness;
  - Skew hashing to guarantee fast lookup for "heavy" buckets;
  - Order of the k-mers induces runs in the weights: suitable for RLE.

- 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.

- Weights add very small extra space and do not impact query time.

- Code in C++17 is available at: https://github.com/jermp/sshash.

(Possible) Future Directions

- Provide an external-memory construction.
  Trade-off RAM usage for disk during construction for better scaling to larger datasets.

- Add support for multi-threading (for queries and construction).

- Add support for other types of queries, like *navigational* queries.

- Use the index as backbone for other problems:
  - positional indexing of k-mers;
  - k-mer quantification across collections of documents;
  - others?

Open Questions

- What happens if we replace the minimizers in SSHash with other types of *seeds*? For example, *strobemers* [Sahlin, 2021], *bi-directional string anchors* [Loukides and Pissis, 2021], …
- What if we change the *hash function* used to select the minimizers?
- Does it lead to an improvement in space (less seeds/lower density)?
- Beyond SPSS: allow duplicates in the representation, e.g., *matchtigs* [Schmidt et al., 2022]?
- What is the cost of *dynamism*, i.e., support for insertions/deletions?

Thank you for the attention!