Minimal Perfect Hashing and K-Mer String Dictionaries

Giulio Ermanno Pibiri
ISTI-CNR

22/12/2021

Agenda

0. About Me (5 minutes)
1. Minimal Perfect Hashing (15 minutes)
2. K-Mer String Dictionaries (15 minutes)

About Me — Short CV

- **Education**
  + Nov. 2015 — Oct. 2018: PhD Degree in Computer Science (INF/01), University of Pisa
  + 2012 — 2014: Master Degree in Computer Science and Networking, University of Pisa
  + 2009 — 2012: Bachelor Degree in Computer Engineering, University of Florence
- **Research Positions — Academic Age: 2017-present (almost 5 years)**
  + Nov. 2018 — present: Postdoc Research Fellow in Computer Science, ISTI-CNR
  + Nov. 2015 — Oct. 2018: PhD Student in Computer Science, University of Pisa

  + 3 european projects
  + 1 collaboration with eBay (California)
- **Visiting Student**
  + May 2018 — Oct. 2018: The University of Melbourne, Melbourne, Australia
  + Apr. 2018: RIKEN AIP, Tokyo, Japan
- **Teaching**
  + April 2022 — Data Compression Course, PhD Program in Ingegneria dell'Informazione, University of Pisa
  + Feb. 2016 — June 2020, University of Pisa
- **Committees**
  + Program: SIGIR'22, ECIR'22, WSDM'22, AICT'21, CIKM'21, SIGIR'21, ECIR'21, WSDM'21, CIKM'20, SIGIR'20, SIGIR'19
  + Organizing: ESA'20, CPM'19, SPIRE'17, SIGIR'16
- **Awards**
  2 x Young Researcher Award (ISTI-CNR), Master Degree Award (Scuola Superiore Sant'Anna), Best Master Thesis (EATCS)

About Me — Research Activity

Keywords: **Data Structures**, **Algorithms**, **Data Compression**, **Indexing**, **Efficiency**

Design compressed data structures and algorithms to **index and search large quantities of data**.

Efficiency is the key to:

+ **build better applications** in terms of reduced latency to access information (enhanced user experience);
+ **save computer resources** (power and storage machines).

About Me — Research Activity

Keywords: **Data Structures**, **Algorithms**, **Data Compression**, **Indexing**, **Efficiency**

Design compressed data structures and algorithms to **index and search large quantities of data**.

Efficiency is the key to:
+ **build better applications** in terms of reduced latency to access information (enhanced user experience);
+ **save computer resources** (power and storage machines).

Some Example Problems

- Inverted Indexes (TOIS'17, WSDM'19, TKDE'19, CSUR'20)
- Language Modeling (SIGIR'17, TOIS'19)
- RDF Triples (TKDE'20)
- Query Auto-Completion (SIGIR'20, collaboration with eBay)
- Prefix-Sums (SPE'20)
- Bitmap Compression (DCC'21)
- Rank/Select Queries (INFOSYS'21)
- Minimal Perfect Hashing (SIGIR'21)

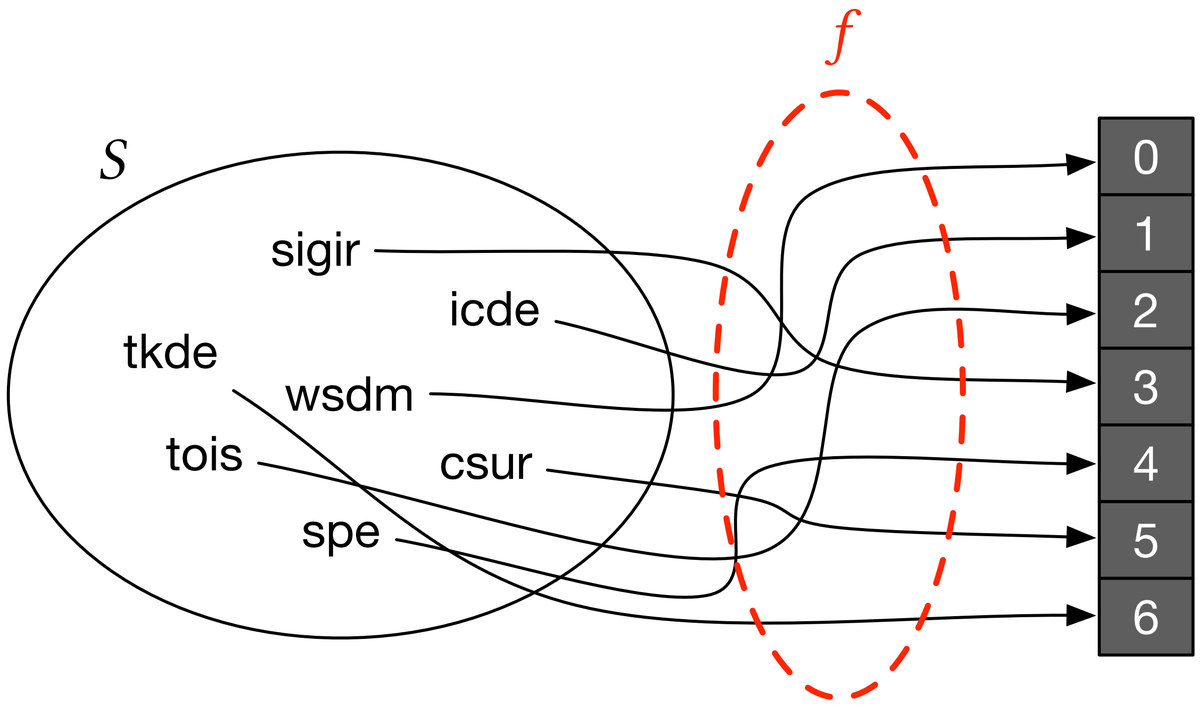
https://github.com/jermp

Part 1
Minimal Perfect Hashing

Minimal Perfect Hashing (MPH)

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* for S.

- **Lower bound of 1.44 bits/key.**
- **Built once and evaluated many times.**
- Many algorithms available, like:
  - FCH (1992)
  - CHD (2009)
  - EMPHF (2014)
  - GOV (2016)
  - BBHash (2017)
  - RecSplit (2019)
  - PTHash (2021)



Context and Motivations

Space-efficient and *fast retrieval* of $\langle key, value \rangle$ pairs from a *static* set.

Some examples:

- Reserved words in programming languages.
- Garbage collectors.
- Command names in interactive systems.
- Lexicon of inverted indexes.
- Indexing of q-grams for language models.
- Web page URLs: DNS, page ranking, ecc.

Indeed MPH: Fast and Compact Immutable Key-Value Stores

Alex Shinn Follow

Feb 4, 2018 · 9 min read

https://medium.com/indeed-engineering/indeed-mph-fast-and-compact-immutable-key-value-stores-302a1cf2eef0

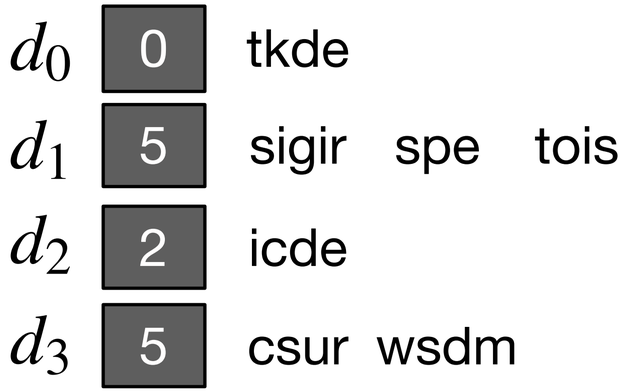
PTHash — Overview

- **Flexibility**: minimal and non-minimal perfect hash functions
- **Space/Time Efficiency**: fast lookup within compressed space
- **External-Memory Scaling**: use disk if not enough RAM is available
- **Parallel Construction**: use more threads to speed up construction
- **Configurable**: can offer different trade-offs (between construction time, lookup time, and space effectiveness)
- **C++ code** available at: https://github.com/jermp/pthash

FCH Construction

Fox, Chen, and Heath, SIGIR'92

- Distribute keys into m *buckets* using hashing and compute a displacement d_i for bucket i such that $f(x) = (h(x) + d_i) \bmod n$, and no collisions occur.
- Use $m = \lceil cn / \log_2 n \rceil$ buckets for n keys and a given parameter c.
- **One memory access per lookup.**



FCH Construction — Remarks

- To guarantee that all positions in the table are tested with *uniform probability*, displacements have to be tried at random → the best we can hope for is $\lceil \log_2 n \rceil$ bits per bucket.
  For $\lceil cn/\log_2 n \rceil$ buckets, it costs cn total bits. Large space for large c.

- Up to n trials to "fit" a pattern.
  If a successful displacement is not found for a bucket → rehash.
  Slow for small c.

  Example. For 10^8 64-bit random keys and $c = 3.0$, FCH takes 1h 10m: other techniques can do the same in 1m or less.

- **Extremely fast lookup.**

PTHash — Intuition

- If the table of displacements were **compressible**, we could afford to use a parameter $c' > c$ and run the search faster, such that the size of the compressed table is $\approx cn$ bits.
- Now, how to achieve compression? Re-design the **search step**.

PTHash — From Displacements to Pilots

$$f(x) = (h(x) + d_i) \bmod n \quad \rightarrow \quad f(x) = (h(x) \oplus h(k_i)) \bmod n$$

PTHash — From Displacements to Pilots

$$f(x) = (h(x) + d_i) \bmod n \quad \rightarrow \quad f(x) = (h(x) \oplus h(k_i)) \bmod n$$

- The bitwise **XOR** between two random fingerprints is another random fingerprint → displacement of keys at random.
- New random patterns generated with every tried pilot, even when pilots are tried **in order**, that is:

  $k_i = 0,1,2,3,\ldots$

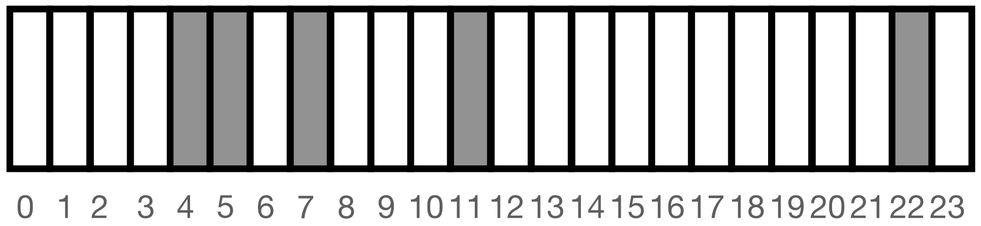
→ **Pilots will be small on average and repetitive, *hence* compressible.**

PTHash — XOR Demo

$x =$ “A View From the Top of the World”

$k_i =$ **0**

$n = 24$



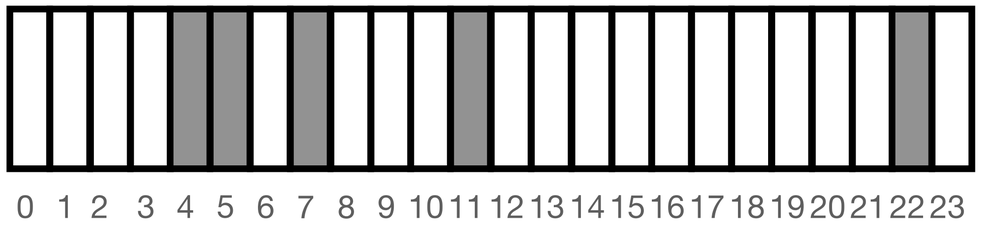
PTHash — XOR Demo

$x =$ “A View From the Top of the World”

$k_i =$ **0**

$h(x) =$ 00011001010111011110010100001111110101011100011011110111000110 11 57.8%

$n = 24$



PTHash — XOR Demo

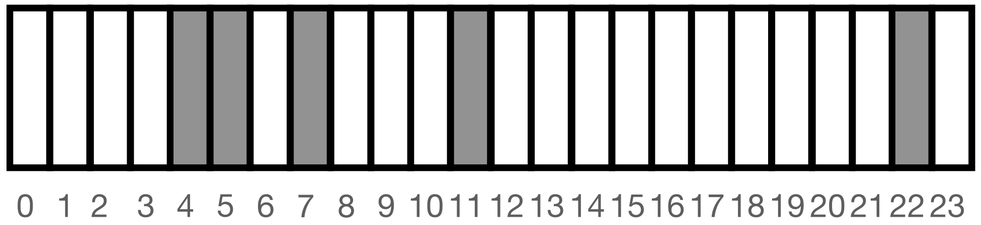
$x =$ “A View From the Top of the World”

$k_i =$ **0**

$h(x) =$ 00011001010111011110010100001111110101011100011011110111000110 11 57.8%

$h(k_i) =$ **10010100001100111101000101110010110110000011100100111110010000 11** **48.4%**

$n = 24$



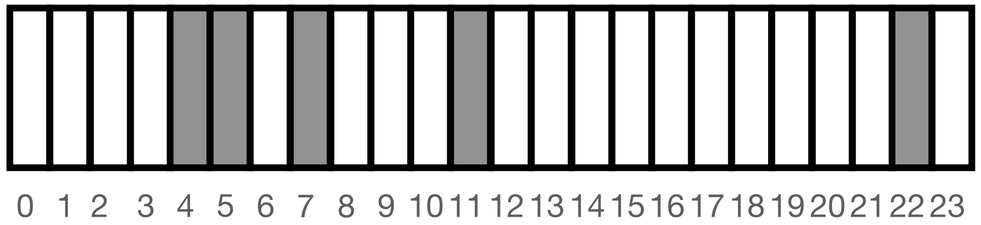
PTHash — XOR Demo

$x =$ “A View From the Top of the World”

$k_i =$ **0**

$h(x) =$	00011001010111011110010100001111110101011100011011110111000110 11	57.8%
$h(k_i) =$	**10010100001100111101000101110010110110000011100100111110010000 11**	**48.4%**
$h(x) \oplus h(k_i) =$	**10001101011011100011010001111101000011011111111111001001010110 00**	**56.2%**

$n = 24$



PTHash — XOR Demo

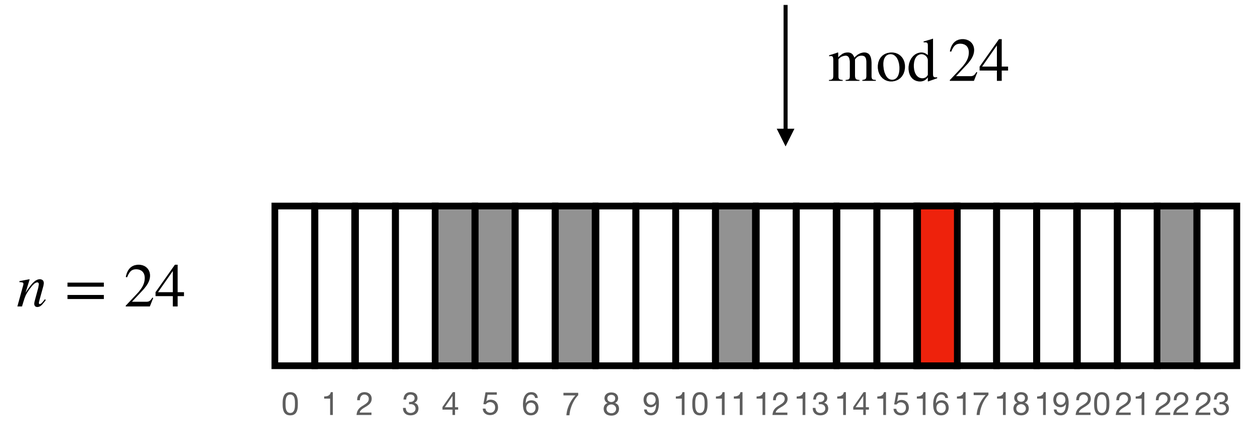
$x =$ “A View From the Top of the World”

$k_i =$ **0**

$h(x) =$ 00011001010111011110010100001111110101011100011011110111000110 11 57.8%

$h(k_i) =$ **10010100001100111101000101110010110110000011100100111110010000 11** **48.4%**

$h(x) \oplus h(k_i) =$ **10001101011011100011010001111101000011011111111111001001010110 00** **56.2%**



PTHash — XOR Demo

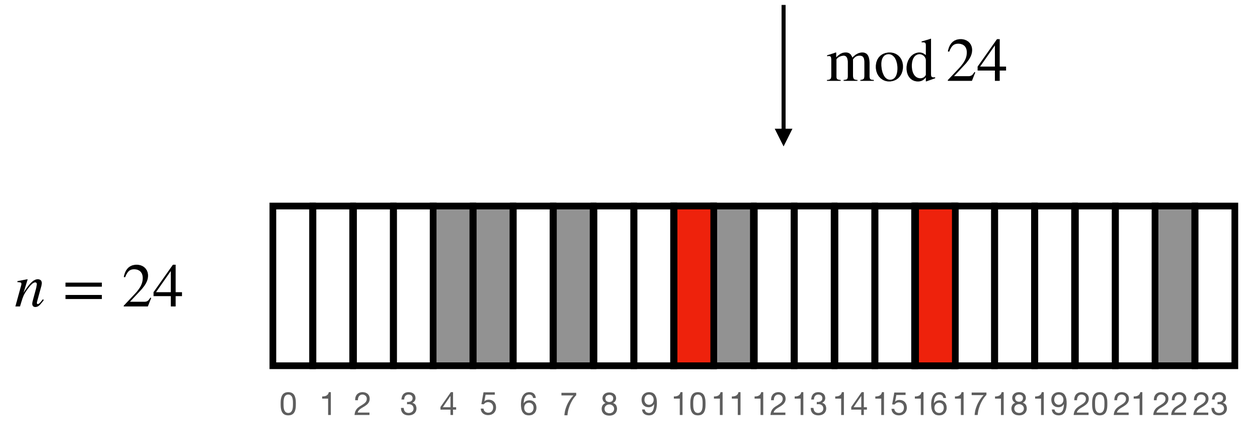
$x =$ “A View From the Top of the World”

$k_i =$ **1**

$h(x) =$ 00011001010111011110010100001111110101011100011011110111000110 11 57.8%

$h(k_i) =$ **01011100010111000010100110011110100010010010111101000100001010 01** **45.3%**

$h(x) \oplus h(k_i) =$ **01000101000000011100110010010001010111001110100110110011001100 10** **43.7%**



PTHash — XOR Demo

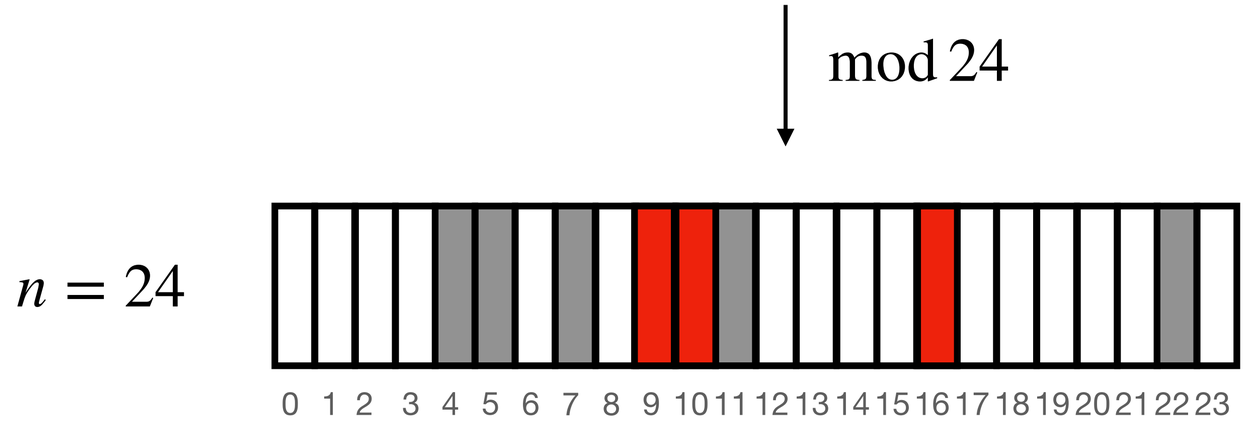
$x =$ “A View From the Top of the World”

$k_i =$ **2**

$h(x) =$ 00011001010111011110010100001111110101011100011011110111000110 11 57.8%

$h(k_i) =$ **00100000101100111111110001111110101001000010110110011101101101010 53.1%**

$h(x) \oplus h(k_i) =$ **00111001111011100001110111110010100111011001110111001100011100 01 57.8%**



PTHash — XOR Demo

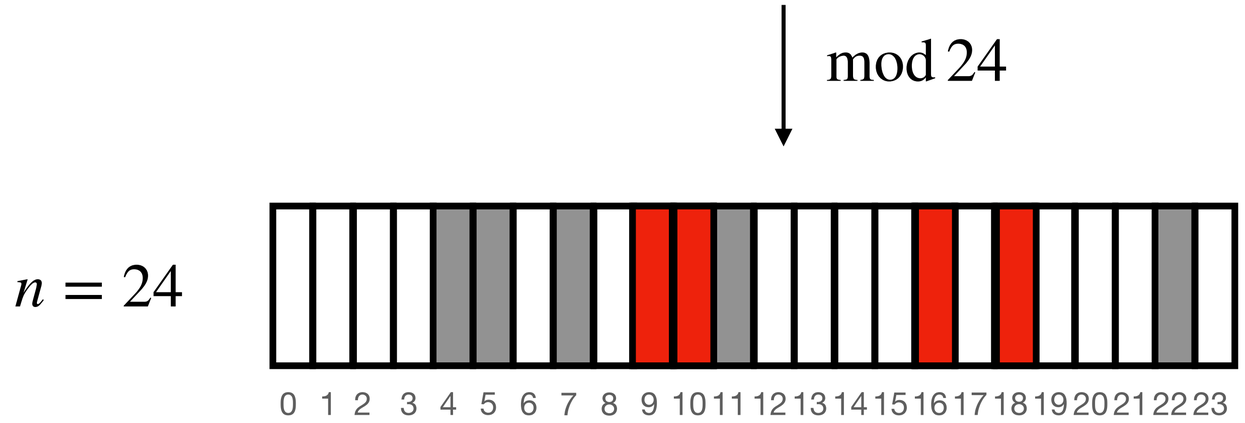
$x =$ “A View From the Top of the World”

$k_i =$ **3**

$h(x) =$ 00011001010111011110010100001111110101011100011011110111000110 11 57.8%

$h(k_i) =$ **01111100100001011000110001101010000101100111011111101010001011001 50.0%**

$h(x) \oplus h(k_i) =$ **01100101110110000110100101100101110000111011000100100011010000 10 43.3%**



PTHash — XOR Demo

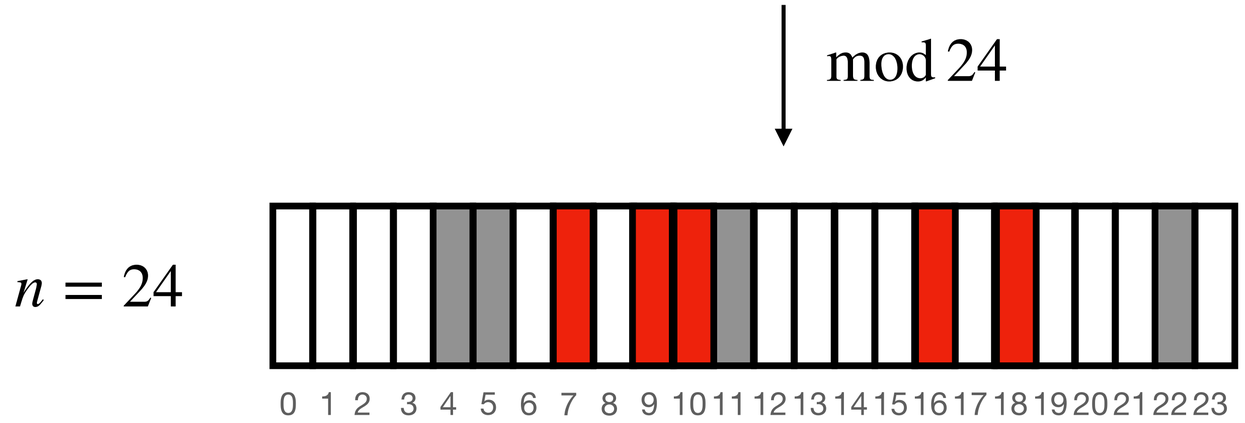
$x =$ “A View From the Top of the World”

$k_i =$ 4

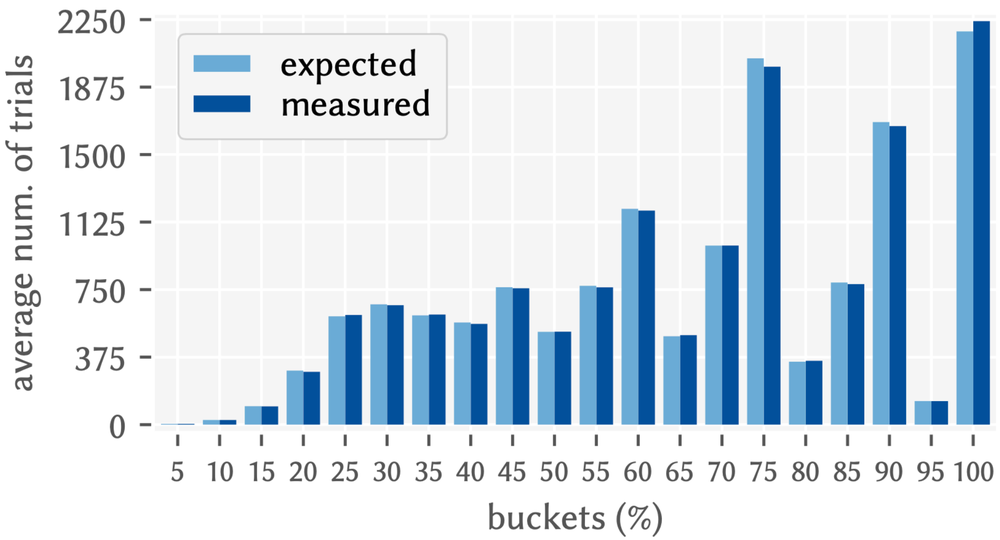
$h(x) =$ 00011001010111011110010100001111110101011100011011110111000110 11 57.8%

$h(k_i) =$ 00001001111011100101110001000001111110111001001000100100000100100 43.8%

$h(x) \oplus h(k_i) =$ **00010000101100111011100101001110001011100101010011010011001111111 51.6%**



PTHash — Entropy



$n = 10^6$ keys, $c = 3.5$ (1.76×10^5 buckets)

$c \rightarrow$	2.5	3.0	3.5	4.0	4.5	5.0	5.5	6.0	6.5	7.0
FCH	16.69	16.85	16.93	16.93	16.87	16.77	16.65	16.49	16.32	16.14
PTHash	13.42	11.68	10.32	9.29	8.48	7.82	7.27	6.82	6.45	6.11

Empirical entropy of the tables, for $n = 10^6$ keys

PTHash — Example

- For 10^8 64-bit random keys and $c = 3.0$ (3 bits/key), FCH takes **1h 10m**.
- PTHash with $\alpha = 0.99$, $c = 6.8$, and **Front-Back Dictionary-based compression** achieves the **same space** (3 bits/key) but builds in **37s** (114×).
- **Both functions evaluate in 35-37 nanosec/key.**

Benchmark with 1B 64-bit random keys

- Processor: Intel i9-9900K @ 3.6 GHz, 32 KiB of L1, 256 KiB of L2 cache
- OS: Ubuntu 20
- Compiler: gcc 9.2.1, with flags `-march=native -O3`
- construction in **internal** memory
- construction is **single-threaded**

Method	$n = 10^9$		
	constr. (secs)	space (bits/key)	lookup (ns/key)
FCH, $c = 3$	—	—	—
FCH, $c = 4$	15904	4.00	35
FCH, $c = 5$	2937	5.00	35
FCH, $c = 6$	2133	6.00	35
FCH, $c = 7$	1221	7.00	35
CHD, $\lambda = 4$	1972	2.17	419
CHD, $\lambda = 5$	5964	2.07	417
CHD, $\lambda = 6$	23746	2.01	416
EMPHF	374	2.61	199
GOV	875	2.23	175
BBHash, $\gamma = 1$	253	3.06	170
BBHash, $\gamma = 2$	152	3.71	143
BBHash, $\gamma = 5$	100	6.87	113
RecSplit, ℓ=5, b=5	233	2.95	220
RecSplit, ℓ=8, b=100	936	1.80	204
RecSplit, ℓ=12, b=9	5700	2.23	197
PTHash			
(i) C-C, α=0.99, c=7	1042	3.23	37
(ii) D-D, α=0.88, c=11	308	3.94	64
(iii) EF, α=0.99, c=6	1799	2.17	101
(iv) D-D, α=0.94, c=7	689	2.99	55

Benchmark with 1B 64-bit random keys

- Processor: Intel i9-9900K @ 3.6 GHz, 32 KiB of L1, 256 KiB of L2 cache
- OS: Ubuntu 20
- Compiler: gcc 9.2.1, with flags `-march=native -O3`
- construction in **internal** memory
- construction is **single-threaded**

Method	$n = 10^9$		
	constr. (secs)	space (bits/key)	lookup (ns/key)
FCH, $c = 3$	—	—	—
FCH, $c = 4$	15904	4.00	35
FCH, $c = 5$	2937	5.00	35
FCH, $c = 6$	2133	6.00	35
FCH, $c = 7$	1221	7.00	35
CHD, $\lambda = 4$	1972	2.17	419
CHD, $\lambda = 5$	5964	2.07	417
CHD, $\lambda = 6$	23746	2.01	416
EMPHF	374	2.61	199
GOV	875	2.23	175
BBHash, $\gamma = 1$	253	3.06	170
BBHash, $\gamma = 2$	152	3.71	143
BBHash, $\gamma = 5$	100	6.87	113
RecSplit, ℓ=5, b=5	233	2.95	220
RecSplit, ℓ=8, b=100	936	1.80	204
RecSplit, ℓ=12, b=9	5700	2.23	197
PTHash			
(i) C-C, α=0.99, c=7	1042	3.23	37
(ii) D-D, α=0.88, c=11	308	3.94	64
(iii) EF, α=0.99, c=6	1799	2.17	101
(iv) D-D, α=0.94, c=7	689	2.99	55

Part 2
K-Mer String Dictionaries*

*Ongoing work!

Problem Definition

K-Mer. A k-mer is a string of length k over the alphabet {A,C,G,T}.
Example: “ACGGTAGAACCGA” is a k-mer of length 13 (k=13).

K-Mer String Dictionary Problem. Given a large string over the alphabet {A,C,G,T} (e.g., a genome or a pan-genome), index all its *distinct* n k-mers, so that the following two operations are supported efficiently:

(1) i = **Lookup**(k-mer), where i ∈ [0,n) if the k-mer belongs to the dictionary
or i = -1 otherwise;

(2) k-mer = **Access**(i).

(plus also iteration and *stateful* membership queries).

Problem Definition

K-Mer. A k-mer is a string of length k over the alphabet {A,C,G,T}.
Example: “ACGGTAGAACCGA” is a k-mer of length 13 (k=13).

K-Mer String Dictionary Problem. Given a large string over the alphabet {A,C,G,T} (e.g., a genome or a pan-genome), index all its *distinct* n k-mers, so that the following two operations are supported efficiently:

(1) i = **Lookup**(k-mer), where i $\in$ [0,n) if the k-mer belongs to the dictionary
or i = -1 otherwise;

(2) k-mer = **Access**(i).

(plus also iteration and *stateful* membership queries).

Like a MPHF but *exact* detection of out-of-set keys.

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

Preliminary Observations

- The algorithmic literature about *(compressed) string dictionaries* is rich of solutions (e.g., Front-Coding, path-decomposed tries, double-array tries, Bonsai, ecc.), but are relevant for "generic strings":
  + variable-length strings;
  + 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 symbols (very low entropy).

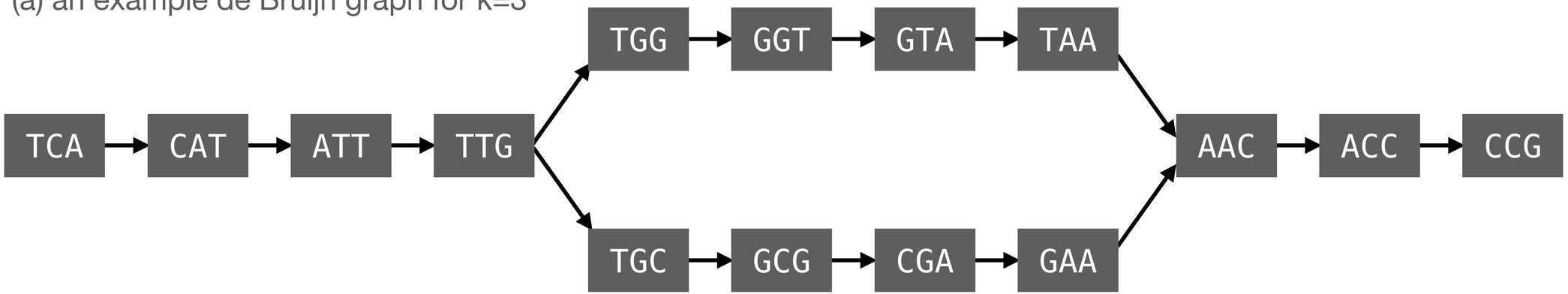
Example for k = 13.

```
ACGGTAGAACCGATTCAAATTCGA …
ACGGTAGAACCGA
 CGGTAGAACCGAT
  GGTAGAACCGATT
   GTAGAACCGATTC
    TAGAACCGATTCA
     AGAACCGATTCAA
      GAACCGATTCAAA
       AACCGATTCAAAT
         …
```

de Bruijn graphs

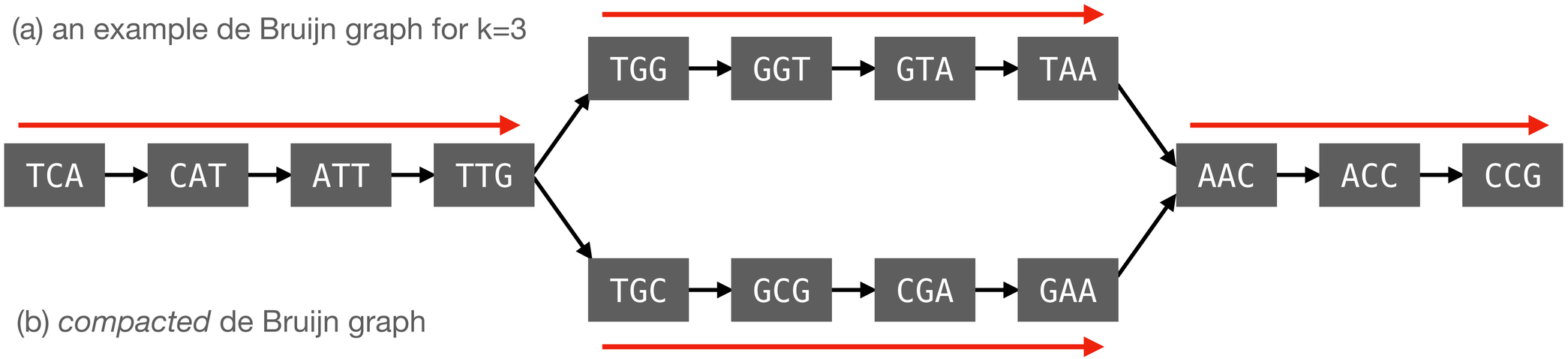
Equivalence between a set of k-mers and a *de Bruijn* graph.

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

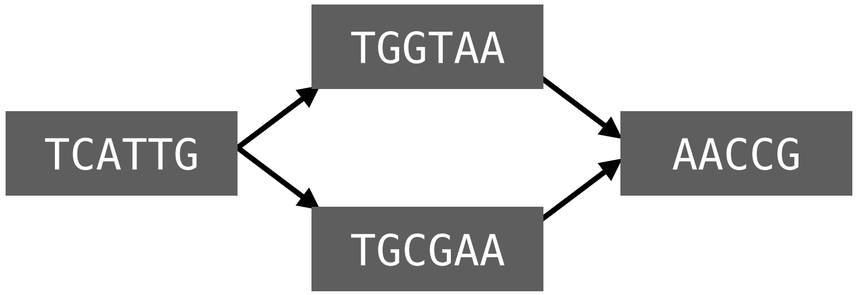


de Bruijn graphs

Equivalence between a set of k-mers and a *de Bruijn* graph.



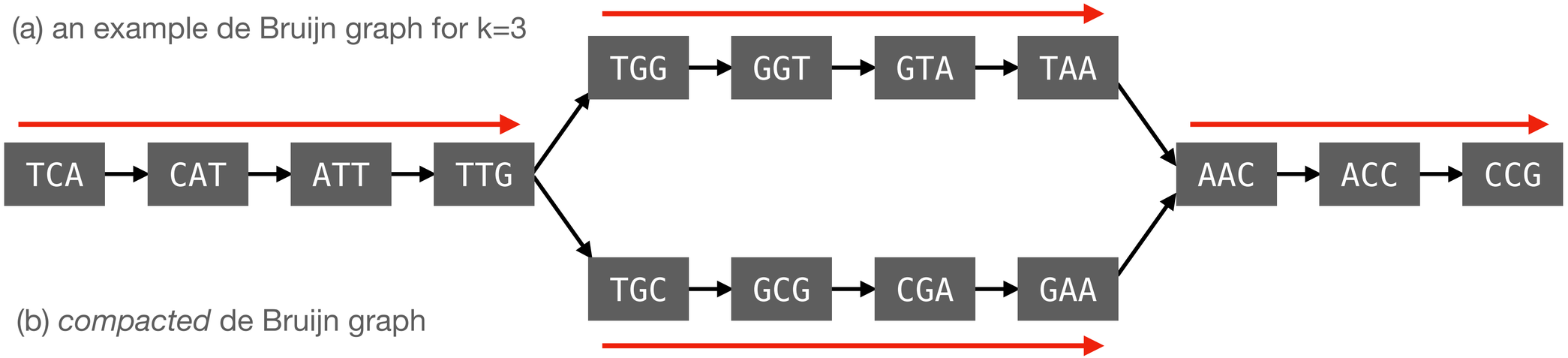
(b) *compacted* de Bruijn graph



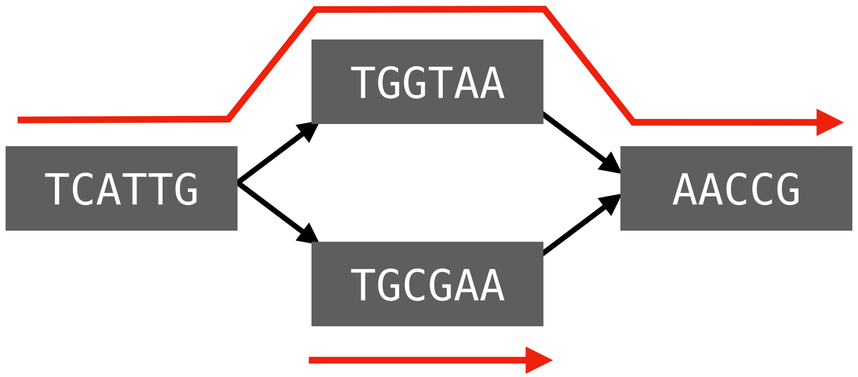
de Bruijn graphs

Equivalence between a set of k-mers and a *de Bruijn* graph.

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



(b) *compacted* de Bruijn graph



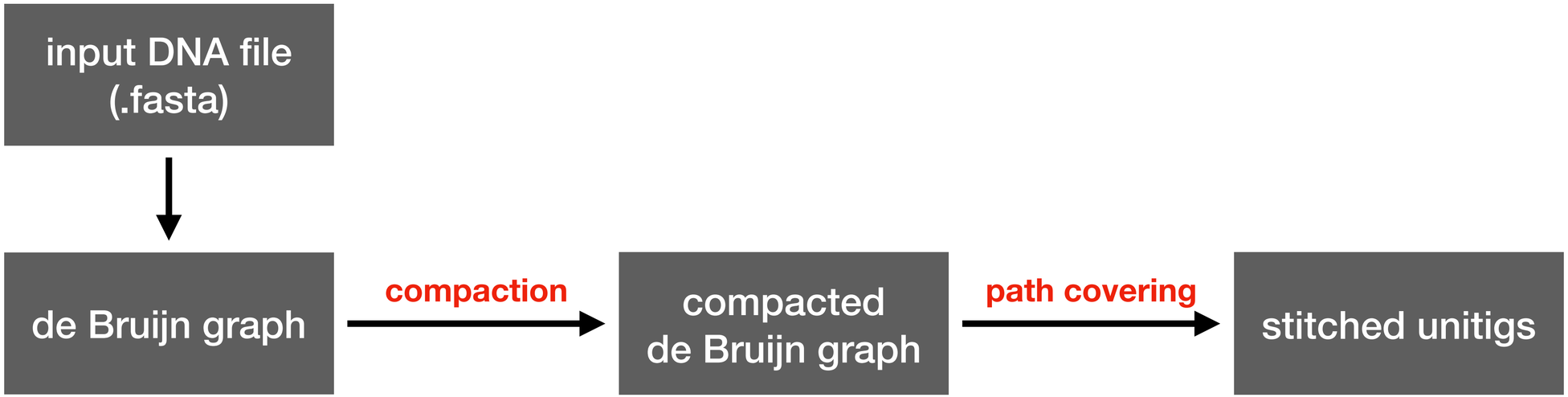
(c) set of *stitched (maximal) unitigs*

TCATTGGTAACCG

TGCGAA

de Bruijn graphs

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



+ no duplicate k-mers
+ heuristic method to minimise the number of symbols

Minimizers

Minimizer. Given a k-mer and an order relation *h*, its *minimizer* of length m < k is its *smallest* m-mer according to the order *h*.

Example. Given “ACGGTAGAACCGA” and m = 4 (k-m+1=13-4+1=10 4-mers):

```
ACGGTAGAACCGA
ACGG
 CGGT
  GGTA
   GTAG
    TAGA
     AGAA
      GAAC
       AACC  <-- If h is the lexicographic order.
        ACCG
         CCGA
```

Super-k-mers

Property. Consecutive k-mers share the same minimizer → there are *far less* minimizers than k-mers ($\approx (k - m + 1)/2$ times less minimizers) → *sparse* indexing.

Example:

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

Super-k-mer. Let a *super-k-mer* be a *maximal* list of consecutive k-mers sharing the same minimizer.

Super-k-mers

Property. Consecutive k-mers share the same minimizer → there are *far less* minimizers than k-mers ($\approx (k - m + 1)/2$ times less minimizers) → *sparse* indexing.

Example:

```
ACGGTAGAACCGATTCAAATTCGA …
ACGGTAGAACCGA
 CGGTAGAACCGAT
  GGTAGAACCGATT
   GTAGAACCGATTC
    TAGAACCGATTCA
     AGAACCGATTCAA
      GAACCGATTCAAA
       AACCGATTCAAAT
         …
```

Super-k-mer. Let a *super-k-mer* be a *maximal* list of consecutive k-mers sharing the same minimizer.

Example: “ACGGTAG**AACC**GATTCAAA” is a super-k-mer from the above string.

Super-k-mers

Observation. A super-k-mer of length s is a **space-efficient** representation of the set of its constituent $s-k+1$ k-mers because it costs $2s/(s-k+1)$ bits/k-mer which is $\approx$ 2 bits/k-mer if s is sufficiently large and/or we have *long chains of super-k-mers.* Much better than $2k$ bits/k-mer.

Example for k=13:
ACGGTAG**AACC**GATTCAAA (2x19=38 bits for 7 k-mers, i.e., 5.43 bits/k-mer vs. 2x13=26 bits/k-mer)

```
ACGGTAGAACCGA
 CGGTAGAACCGAT
  GGTAGAACCGATT
   GTAGAACCGATTC
    TAGAACCGATTCA
     AGAACCGATTCAA
      GAACCGATTCAAA
```

How to index super-k-mers?

Our Solution — Overview

Idea 1. *Conceptually*, partition the unitigs into **super-k-mers** by a left-to-right parse of the unitigs. Call a **bucket** for a minimizer the set of its super-k-mers. To lookup a k-mer, we will inspect the bucket of its minimizer.

Idea 2. Do **not** break the unitigs as to *preserve the chains of super-k-mers* (every time we break a string, we pay 2(k-1) bits): build an **inverted index** on minimizers to locate super-k-mers in the unitigs.

Our Solution — Overview

Idea 1. *Conceptually*, partition the unitigs into **super-k-mers** by a left-to-right parse of the unitigs. Call a **bucket** for a minimizer the set of its super-k-mers. To lookup a k-mer, we will inspect the bucket of its minimizer.

Idea 2. Do **not** break the unitigs as to *preserve the chains of super-k-mers* (every time we break a string, we pay 2(k-1) bits): build an **inverted index** on minimizers to locate super-k-mers in the unitigs.

Example for k=13 and m=4:

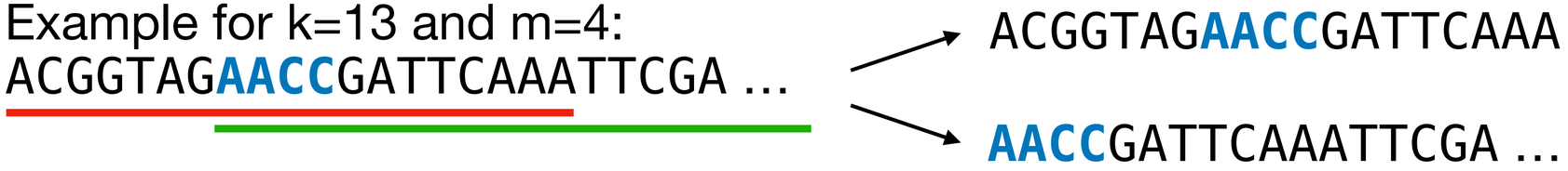
ACGGTAG**AACC**GATTCAAATTCGA …

Our Solution — Overview

Idea 1. *Conceptually*, partition the unitigs into **super-k-mers** by a left-to-right parse of the unitigs. Call a **bucket** for a minimizer the set of its super-k-mers. To lookup a k-mer, we will inspect the bucket of its minimizer.

Idea 2. Do **not** break the unitigs as to *preserve the chains of super-k-mers* (every time we break a string, we pay 2(k-1) bits): build an **inverted index** on minimizers to locate super-k-mers in the unitigs.

Example for k=13 and m=4:



Idea 3. Exploit the *skew* distribution of minimizers to keep good space effectiveness and make searches very fast.

Super-k-mer Partitioning + Inverted Index

Input: some (stitched) unitigs

```
>
AGATGATGAACCTGAAAACATCCTGAAAATCGTCAA
AGAATGGCGGCGTTCACAGGGGCTACCCTTGTTTAA
AGACTCTAAATAAAGTA
>
ATTTTCAGGATGTTTTCAGGTTCATCATCTCCCTTC
TTTGCAGGATAGTAGATAAGATCGCTCATCAACGGA
TGTTGTGTAATTCTGGTAAGATGTTCTTCTAGATCA
TCCCAATATTTGTCAAGCACTTCCCCTTTTAATTGA
GCGTTATCCCCGG
>
AGATGATGAACCTGAAAACATCCTGAAAATTGTCAA
AGAATGGCGGCGTTCACAGGGGCTA
>
ATTGTCAAAGAATGGCGGCGTTCACAGGGGTTACCC
TTGTTTAAAGACTCTAAATAAAGTAGATAATAAAAC
TATATATGGAACATCATCGCATCTGG
```

Super-k-mer Partitioning + Inverted Index

Input: some (stitched) unitigs

```
0
AGATGATGAACCTGAAAACATCCTGAAAATCGTCAA
AGAATGGCGGCGTTCACAGGGGCTACCCTTGTTTAA
AGACTCTAAATAAAGTA
>
ATTTTCAGGATGTTTTCAGGTTCATCATCTCCCTTC
TTTGCAGGATAGTAGATAAGATCGCTCATCAACGGA
TGTTGTGTAATTCTGGTAAGATGTTCTTCTAGATCA
TCCCAATATTTGTCAAGCACTTCCCCTTTTAATTGA
GCGTTATCCCCGG
>  246
AGATGATGAACCTGAAAACATCCTGAAAATTGTCAA
AGAATGGCGGCGTTCACAGGGGCTA
>
ATTGTCAAAGAATGGCGGCGTTCACAGGGGTTACCC
TTGTTTAAAGACTCTAAATAAAGTAGATAATAAAAC
TATATATGGAACATCATCGCATCTGG
```

Inverted index + MPHF

```
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
```

- k=31 and m=8:
  285 k-mers and 24 minimizers
- **Lookup**(k-mer):
  1. Calculate minimizer
  2. Retrieve its super-k-mers from the inverted index
  3. Search the k-mer (scan **at most** k-m+1 k-mers)

  Example:
  GAACCTGAAAACATCCTGAAAATTGTCAAAG

Super-k-mer Partitioning + Inverted Index

Input: some (stitched) unitigs

```
 0
AGATGATGAACCTGAAAACATCCTGAAAATCGTCAA
AGAATGGCGGCGTTCACAGGGGCTACCCTTGTTTAA
AGACTCTAAATAAAGTA
>                                    119
ATTTTCAGGATGTTTTCAGGTTCATCATCTCCCTTC
TTTGCAGGATAGTAGATAAGATCGCTCATCAACGGA
TGTTGTGTAATTCTGGTAAGATGTTCTTCTAGATCA
TCCCAATATTTGTCAAGCACTTCCCCTTTTAATTGA
GCGTTATCCCCGG
>   246
AGATGATGAACCTGAAAACATCCTGAAAATTGTCAA
AGAATGGCGGCGTTCACAGGGGCTA
>
ATTGTCAAAGAATGGCGGCGTTCACAGGGGTTACCC
TTGTTTAAAGACTCTAAATAAAGTAGATAATAAAAC
TATATATGGAACATCATCGCATCTGG
```

Inverted index + MPHF

```
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
```

- k=31 and m=8:
  285 k-mers and 24 minimizers
- **Lookup**(k-mer):
  1. Calculate minimizer
  2. Retrieve its super-k-mers from the inverted index
  3. Search the k-mer (scan **at most** k-m+1 k-mers)

Example:

```
GAACCTGAAAACATCCTGAAAATTGTCAAAG
GATAGTAGATAAGATCGCTCATCAACGGATG
```

Skew Hashing

Property. Minimizers have a (very) *skew* distribution for sufficiently large m.

Skew Hashing

Property. Minimizers have a (very) *skew* distribution for sufficiently large m.

Example on the 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! (buckets of size 1)

Skew Hashing

Property. Minimizers have a (very) *skew* distribution for sufficiently large m.

Example on the 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! (buckets of size 1)

Idea 3. Choose a threshold T and build a MPHF for all the k-mers belonging to buckets of size > T. If a k-mer belongs to a bucket of size s, then we only spend $\lceil \log_2 s \rceil$ bits/k-mer to indicate its super-k-mer.
We only pay the cost for the MPHF for a *small* fraction of the total k-mers
but guarantee that each lookup scans *at most* T super-k-mers.

Skew Hashing

Property. Minimizers have a (very) *skew* distribution for sufficiently large m.

Example on the 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! (buckets of size 1)

Idea 3. Choose a threshold T and build a MPHF for all the k-mers belonging to buckets of size > T. If a k-mer belongs to a bucket of size s, then we only spend $\lceil \log_2 s \rceil$ bits/k-mer to indicate its super-k-mer.
We only pay the cost for the MPHF for a *small* fraction of the total k-mers but guarantee that each lookup scans *at most* T super-k-mers.

Continuing the example above: **For T = 4, only 0.64% of buckets of size > 4.**
num_kmers belonging to buckets of size > 4 and <= 8: 44487258 (num_bits_per_pos = 3)
num_kmers belonging to buckets of size > 8 and <= 16: 35434987 (num_bits_per_pos = 4)
num_kmers belonging to buckets of size > 16 and <= 32: 29555882 (num_bits_per_pos = 5)
…
num_kmers belonging to buckets of size > 1024 and <= 2048: 4430407 (num_bits_per_pos = 11)
num_kmers belonging to buckets of size > 2048 and <= 4096: 2764374 (num_bits_per_pos = 12)
num_kmers belonging to buckets of size > 4096 and <= **36256**: 2923517 (num_bits_per_pos = 16)
num_kmers 194505733 (7.76%)

Skew Hashing — Example

If the bucket for minimizer “**AACCTGAA**” has size 7 and is

```
2355 81591 83343 99911 110108 120991 281301
0    1     2     3     4      5      6
```

and the kmer “G**AACCTGAA**AACATCCTGAAAATTGTCAAAG” is present in the super-k-mer of position 281301, then it will be costly to scan all the 7 super-k-mers.

Thus, we lookup the k-mer in a MPHF f.

If f(“G**AACCTGAA**AACATCCTGAAAATTGTCAAAG”) = p, then we store 6 in a **compact** array A at position p.

The array A is **compact** because it stores integers v that are $2^2 \leq v < 2^3$, thus requiring $\lceil \log_2 2^3 \rceil = 3$ bits each.

Some Experiments — Setting

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

Dataset	Number of distinct k-mers (k = 31)
Human Chr 14	82,465,393
Human Chr 1+2+3	595,463,222
Whole Human	2,505,445,761

Space, Lookup, and Access

Dataset	Minimizer size (m)	Avg. bits/k-mer		Avg. Lookup time in nanosec		Avg. Access time in nanosec
		without skew index	with skew index	without skew index	with skew index	
Human Chr 14	17	6.25	6.74	1106	594	295
Human Chr 1+2+3	20	7.67	8.03	1753	928	440
Whole Human	20	8.07	8.58	3680	1209	615

- The skew hash-based index grants **2-3X better (average) lookup** time for only **+0.5** bits/k-mer.
- The impact grows for larger datasets.

Comparison against the FM-index

- The FM-index (Ferragina, Manzini, 2000) is a full-text index based on the Burrows-Wheeler transform of the text.

- Massive impact in bioinformatics. Can be built on the stitched unitigs to provide membership data structures for k-mer sets (Chikhi et al., 2014; Rahman, Medvedev, 2021).
  C++ code: https://github.com/jts/dbgfm

Dataset	Avg. bits/k-mer		Avg. Lookup time in nanosec	
	without skew index	with skew index	without skew index	with skew index
Human Chr 14	6.25	6.74	1106	594
Human Chr 1+2+3	7.67	8.03	1753	928

(a) Our

Dataset	Avg. bits/k-mer		Avg. Lookup time in nanosec	
	sampling rate 32	sampling rate 64	sampling rate 32	sampling rate 64
Human Chr 14	5.94	4.22	6601	8999
Human Chr 1+2+3	6.03	4.27	8348	11859

(b) FM-index

Comparison against Blight and Pufferfish

- **Blight** (Marchet et al., 2021) and **Pufferfish** (Almodaresi et al., 2018) are two recent indexes for k-mer sets optimized for *stateful membership queries.*
  C++ code:
  https://github.com/Malfoy/Blight
  https://github.com/COMBINE-lab/pufferfish

- Benchmark: for the whole Human genome, query all k-mers from SRA accessions.
  + SRR5833294 — 34,129,891 reads, each of length 76 bases, 91.65% hits;
  + SRR5901135 — 4,628,576 reads, variable length, 0.002% hits.

Index	Avg. bits/k-mer	SRR5833294 (total time in minutes)	SRR5901135 (total time in minutes)
Blight dense	20.17	11.4	12.2
Blight sparse (b=4)	16.11	26.3	35.9
Pufferfish dense	45.04	10.6	5.8
Pufferfish sparse (e=2)	26.10	16.3	9.8
Our	**10.13**	**6.3**	**2.6**

Thank you for the attention!

References

- Giulio Ermanno Pibiri and Roberto Trani. "*PTHash: Revisiting FCH Minimal Perfect Hashing*". In Proceedings of the 44th International Conference on Research and Development in Information Retrieval (SIGIR). 2021.
- Giulio Ermanno Pibiri and Roberto Trani. "*Parallel and External-Memory Construction of Minimal Perfect Hash Functions with PTHash*". ArXiv. 2021
- Chikhi, Rayan, Antoine Limasset, Shaun Jackman, Jared T. Simpson, and Paul Medvedev. "On the representation of de Bruijn graphs." In *International conference on Research in computational molecular biology*, pp. 35-55. Springer, Cham, 2014.
- Ferragina, Paolo, and Giovanni Manzini. "Opportunistic data structures with applications." In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pp. 390-398. IEEE, 2000.
- Rahman, Amatur, and Paul Medevedev. "Representation of k-mer sets using spectrum-preserving string sets." *Journal of Computational Biology* 28, no. 4 (2021): 381-394.
- Marchet, Camille, Mael Kerbiriou, and Antoine Limasset. "BLight: Efficient exact associative structure for k-mers." *Bioinformatics* (2021).
- Almodaresi, Fatemeh, Hirak Sarkar, Avi Srivastava, and Rob Patro. "A space and time-efficient index for the compacted colored de Bruijn graph." *Bioinformatics* 34, no. 13 (2018): i169-i177.