Indexing the (compacted) colored de Bruijn graph

Scaling up fast reference-based indices

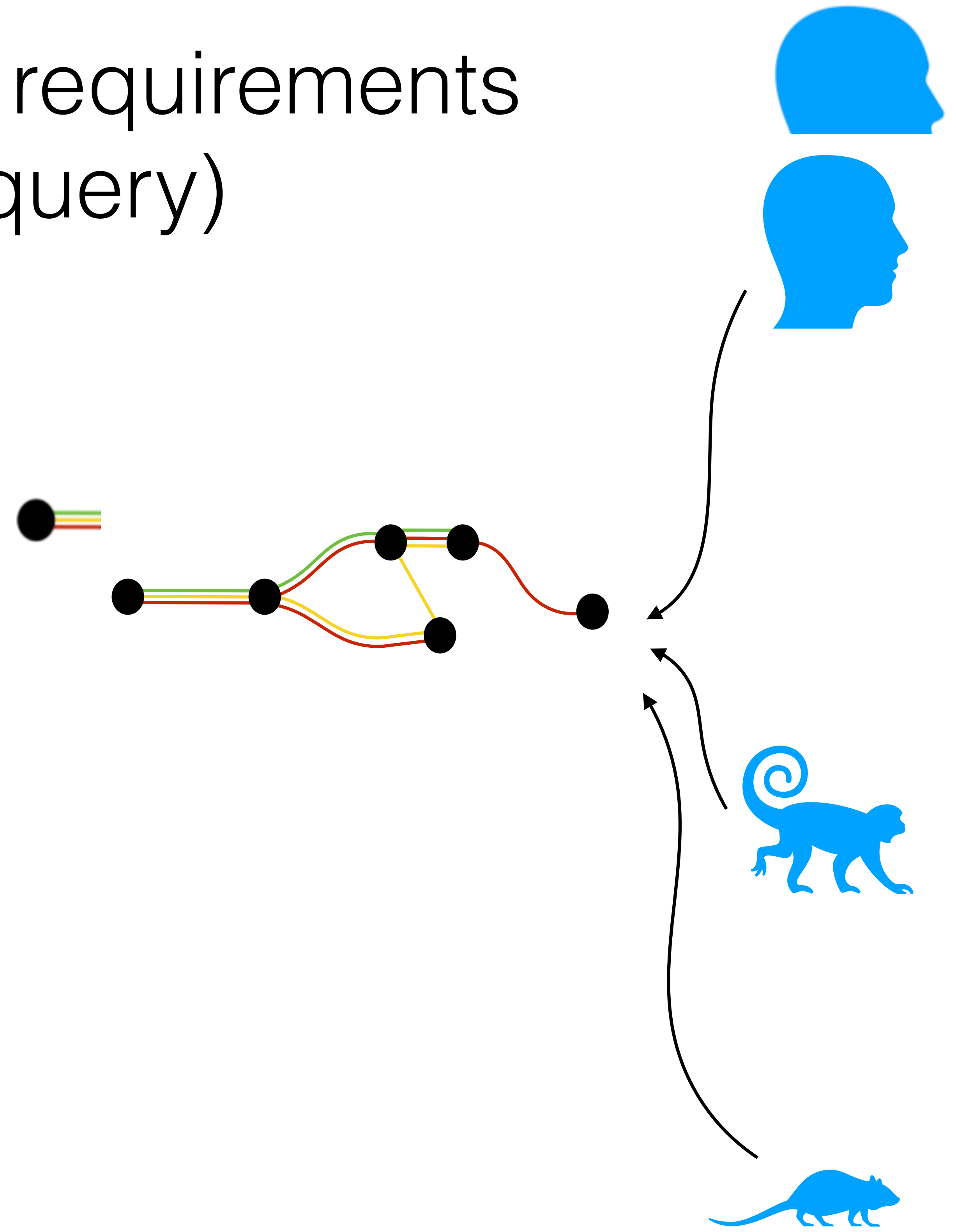
Motivation: Indices used in “ultra-fast” mapping approaches are typically very memory hungry. This is **OK** for transcriptome mapping, but **not scalable** to genomic, metagenomic, pangenomic or population mapping.

Goal: Develop an index with practical memory requirements that maintains the desirable performance (i.e. query) characteristics of the “ultra-fast” indices.

Compacted colored de Bruijn graph (ccdBG)

Built over 1 or more genomes / sequence collections

Index makes use of minimum perfect hashing succinct bit vector representations and (optionally) a new sampling scheme



Pufferfish: An efficient index for the ccdBG

Bioinformatics, 34, 2018, i169–i177
doi: 10.1093/bioinformatics/bty292
ISMB 2018

OXFORD

A space and time-efficient index for the compacted colored de Bruijn graph

Fatemeh Almodaresi†, Hirak Sarkar†, Avi Srivastava and Rob Patro*

Department of Computer Science, Stony Brook University, Stonybrook, NY 11794, USA

Appeared at **ISMB 2018**

- The past decade has largely been dominated by SA/BWT/FM-index-based approaches to reference sequence indexing (e.g. Bowtie, BWA, BWA-MEM, Bowtie2, STAR, etc.)
- There has been a renaissance of sorts for hash-based indexing (deBGA, Brownie, kallisto, mashmap, minimap & minimap2, etc.)
- Pufferfish goes the hashing-based route; *with a twist*.
- Not considering generalized path indices on general seq (e.g. GCSA2 (VG), HISAT2). Interesting, but a different problem.

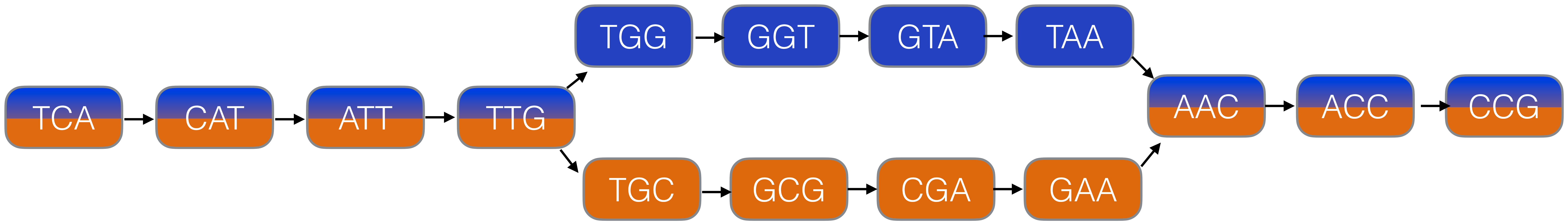
https://github.com/COMBINE-lab/pufferfish

Recall the “colored” de Bruijn Graph

Nodes are k-mers (here k=3)

Edges exist between nodes that overlap by k-1 (in the input)*

Colors encode “origin” of k-mers (e.g., references where they exist)



compacted colored de Bruijn graph

Example from : https://algolab.files.wordpress.com/2016/10/chikhi-milan-18nov.pdf

There are multiple related (but distinct) definitions of the dBG in practice. We adopt the **edge-explicit** version.

The compacted colored dBG as a sequence index

- **Key idea**: represent a collection of sequences using the colored de Bruijn graph (dBG) (Iqbal '12).

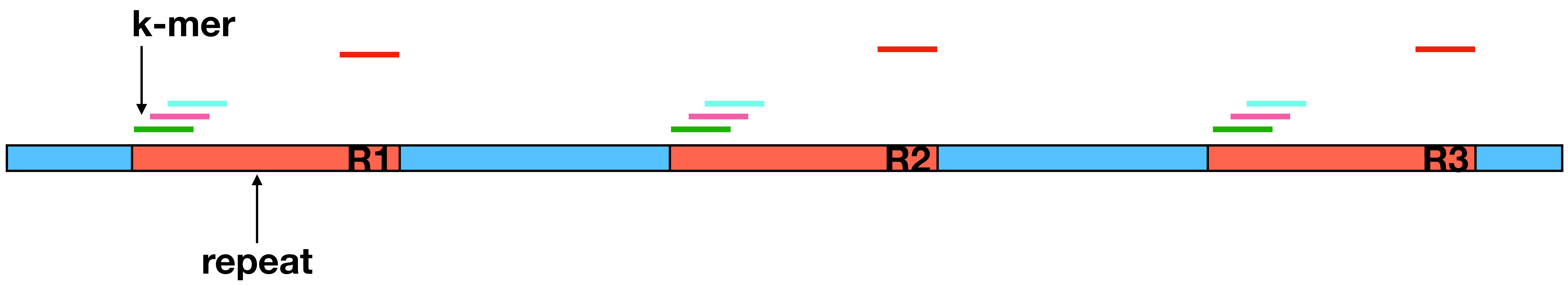
- Each color is an input reference (e.g. genome or transcript).

- Use the compacted colored dBG as an index for reference-based sequence search.

- Redundant sequences (repeats) are implicitly collapsed. **Why is this potentially *much* better than a naive hash?**

The compacted colored dBG as a sequence index

- Redundant sequences (repeats) are implicitly collapsed. **Why is this potentially *much* better than a naive hash?**



List all occurrences individually

— → **R1-l1, R2 - l1, …, RM - l1**

— → **R1-l1+1, R2 - l1+1, …, RM - l1+1**

— → **R1-l1+2, R2 - l1+2, …, RM - l1+2**

⋮ **…**

— → **R1-k, R2 - k, …, RM - k**

Factors out long repeat (k-mer pos always same)

▬ → **R1-l1, R2 - l1, …, RM - l1**

— → **0**

— → **1**

— → **2**

⋮

— → **l1-k**

The cdBG removes redundancy by providing an extra level of indirection

The compacted colored dBG as a sequence index

- Redundant sequences (repeats) are implicitly collapsed. **Why is this potentially *much* better than a naive hash?**

Still, the biggest problem for these schemes, in practice, is *memory usage*

The main culprit is the hash table itself

The cdBG removes redundancy by providing an extra level of indirection

Recall: Minimum Perfect Hashing

Minimum Perfect Hash Function (MPHF)

$$\mathscr{K} \subseteq \mathscr{U}, \quad f : \mathscr{K} \rightarrow \mathbb{N}^{+}$$

if $x \in \mathscr{K}$ **then** $f(x) \in [1, |\mathscr{K}|]$

if $x \in \mathscr{U} \backslash \mathscr{K}$ **then** $f(x) \in [1, |\mathscr{U}|]$ (Like “false positives”)

f is a complete, injective function from $\mathscr{K} \rightarrow [1, |\mathscr{K}|]$

Best methods achieve ~2.1 bits/key ***regardless of key size***

Use BBHash :)

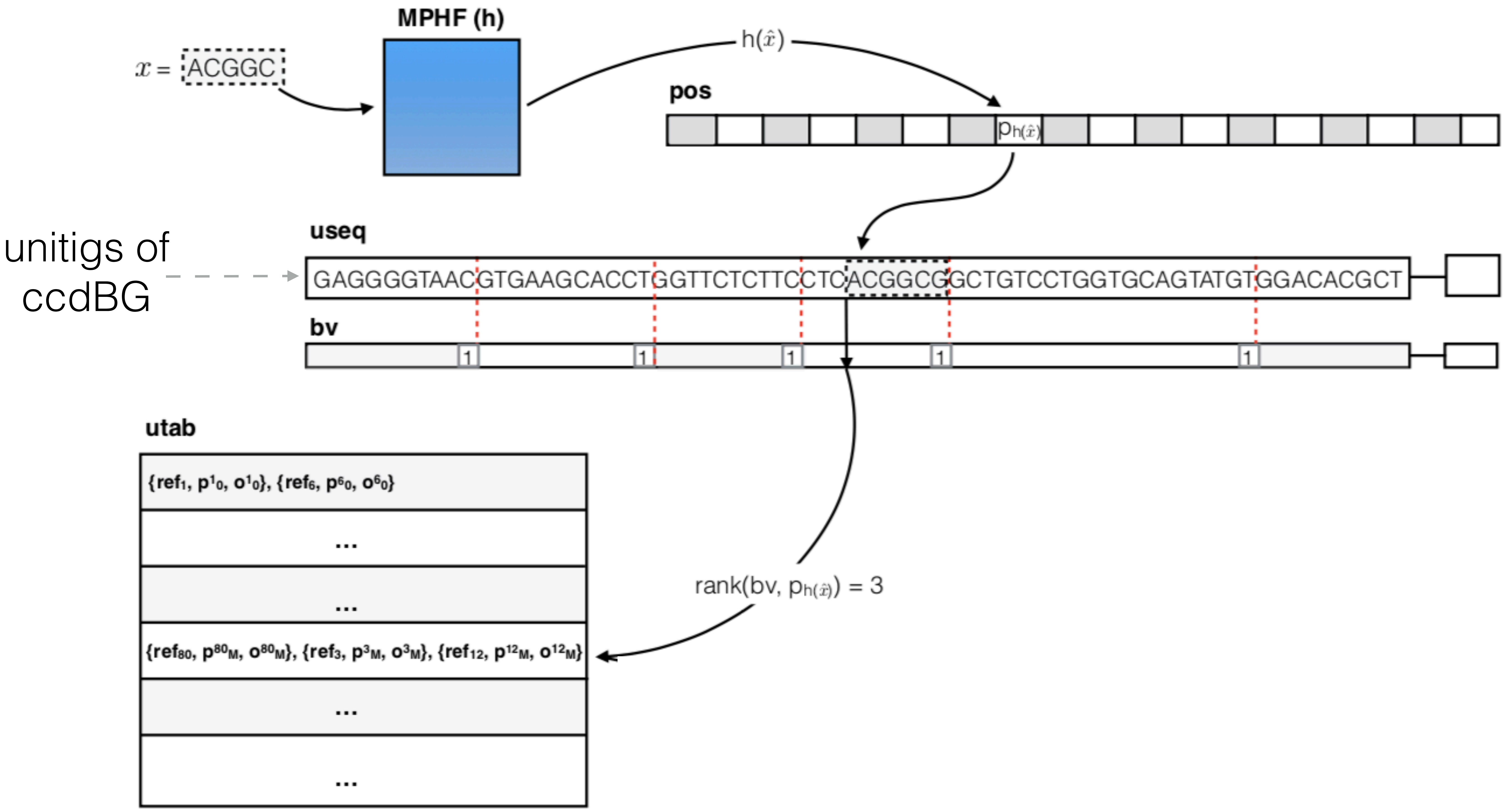
Fast and scalable minimal perfect hashing for massive key sets

Antoine Limasset[1], Guillaume Rizk[1], Rayan Chikhi[2], and Pierre Peterlongo[1]

1 IRISA Inria Rennes Bretagne Atlantique, GenScale team, Campus de Beaulieu 35042 Rennes, France
2 CNRS, CRIStAL, Université de Lille, Inria Lille - Nord Europe, France

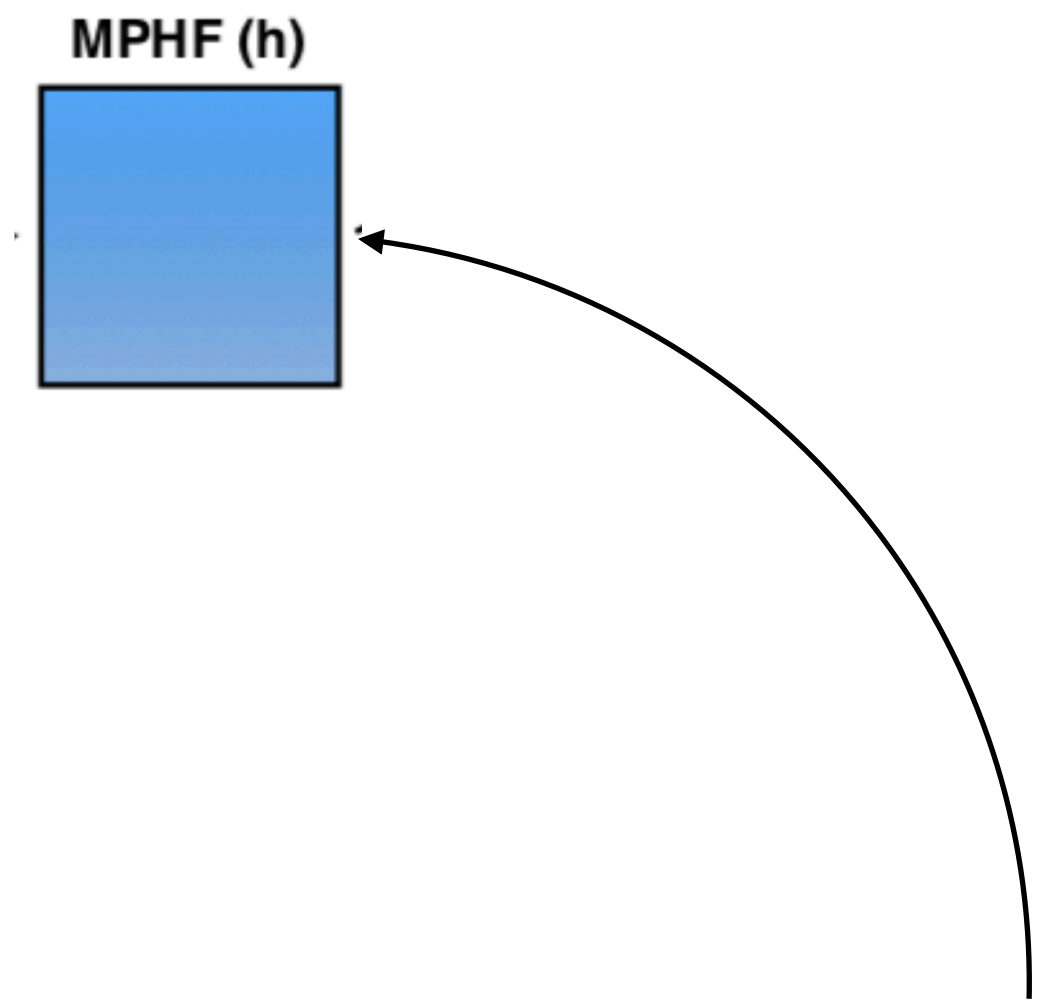
https://github.com/rizkg/BBHash

The **dense** Pufferfish index



Optionally: explicit edge table, equivalence class table

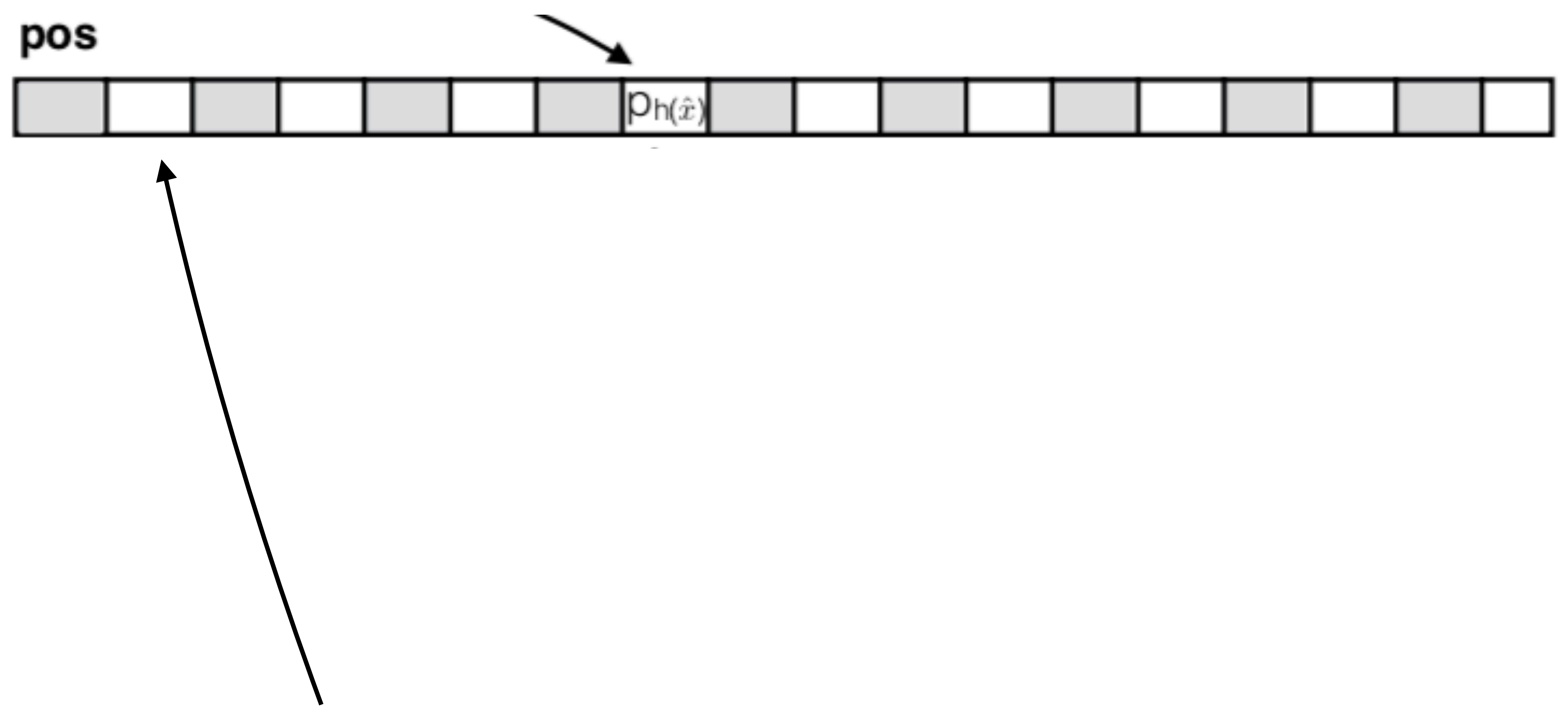
The **dense** Pufferfish index



Maps each valid k-mer to some number in [0,N)

Optionally: explicit edge table, equivalence class table

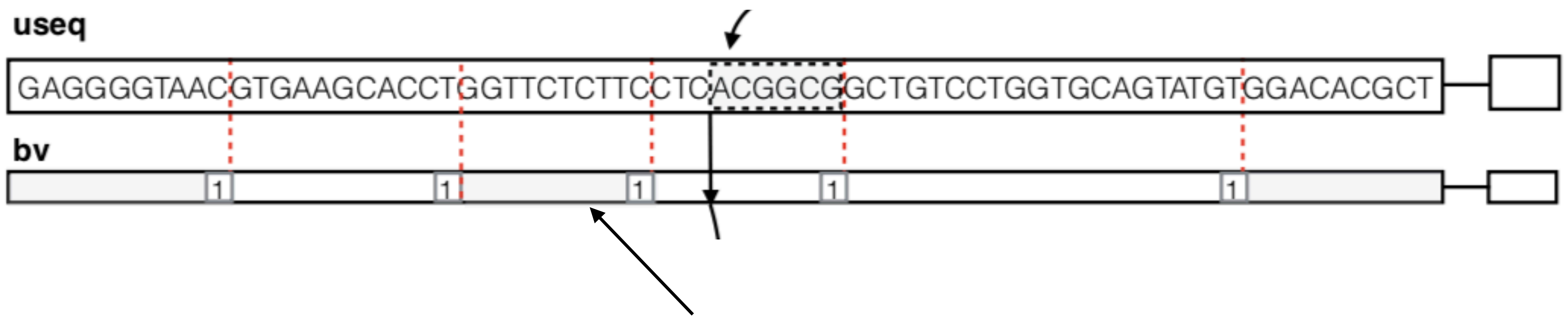
The **dense** Pufferfish index



At index h(x), this table contains the position, in the list of unitigs, of this k-mer

Optionally: explicit edge table, equivalence class table

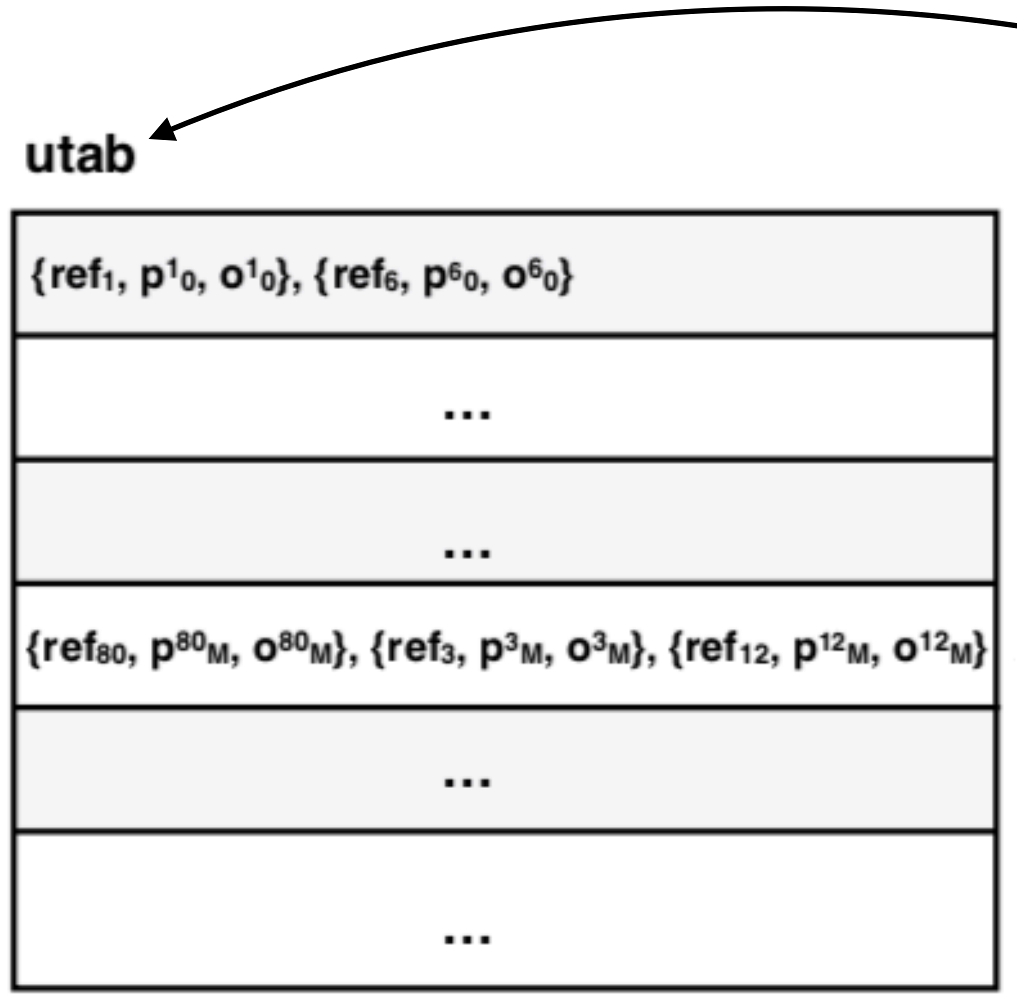
The **dense** Pufferfish index



- **useq contains the uniting sequences concatenated together**
- **bv is a boundary vector that records a 1 at the end of each uniting, and a 0 elsewhere**

Optionally: explicit edge table, equivalence class table

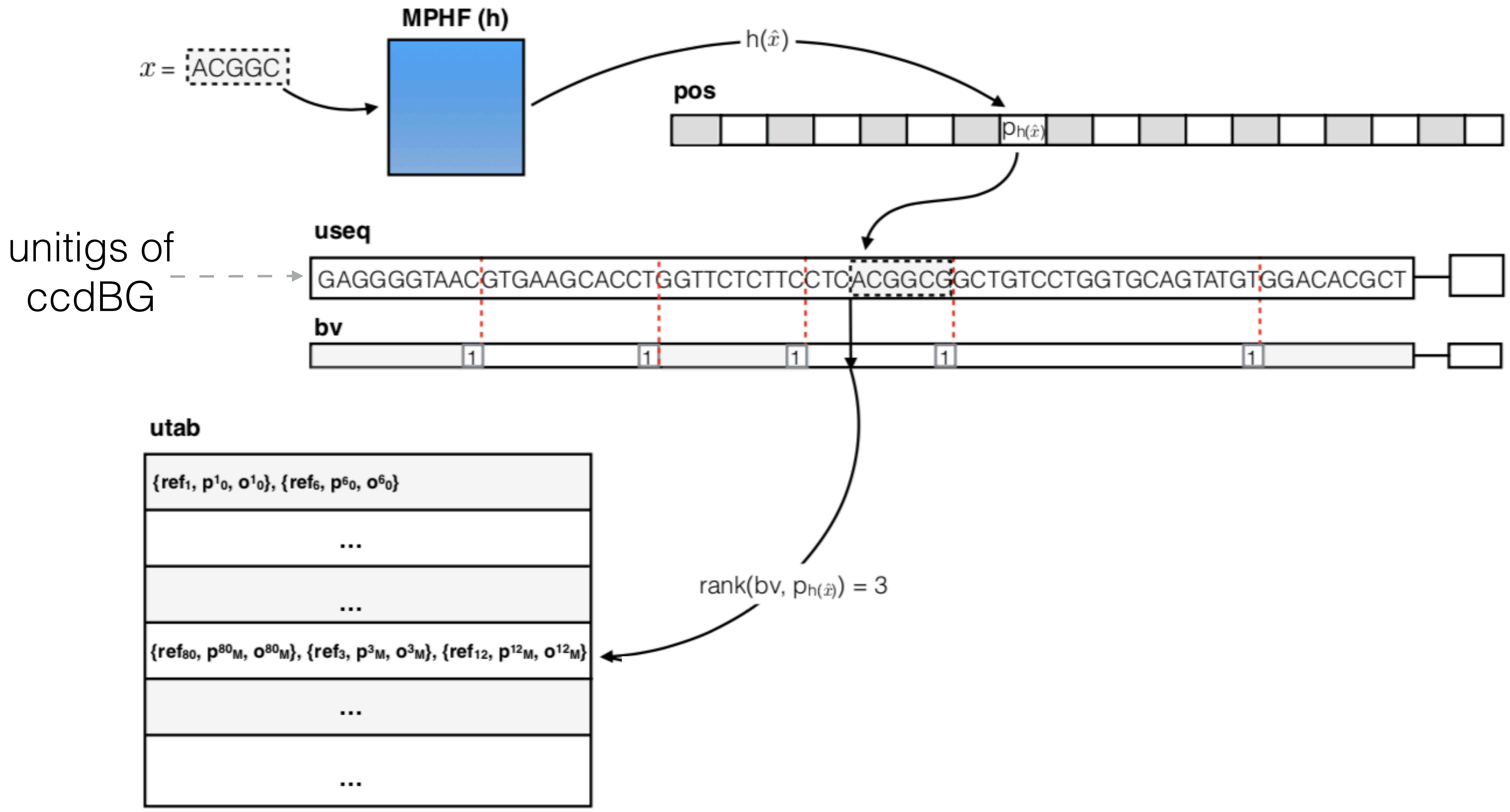
The **dense** Pufferfish index



Records, for each uniting, the list of references, positions and orientations in which it occurs

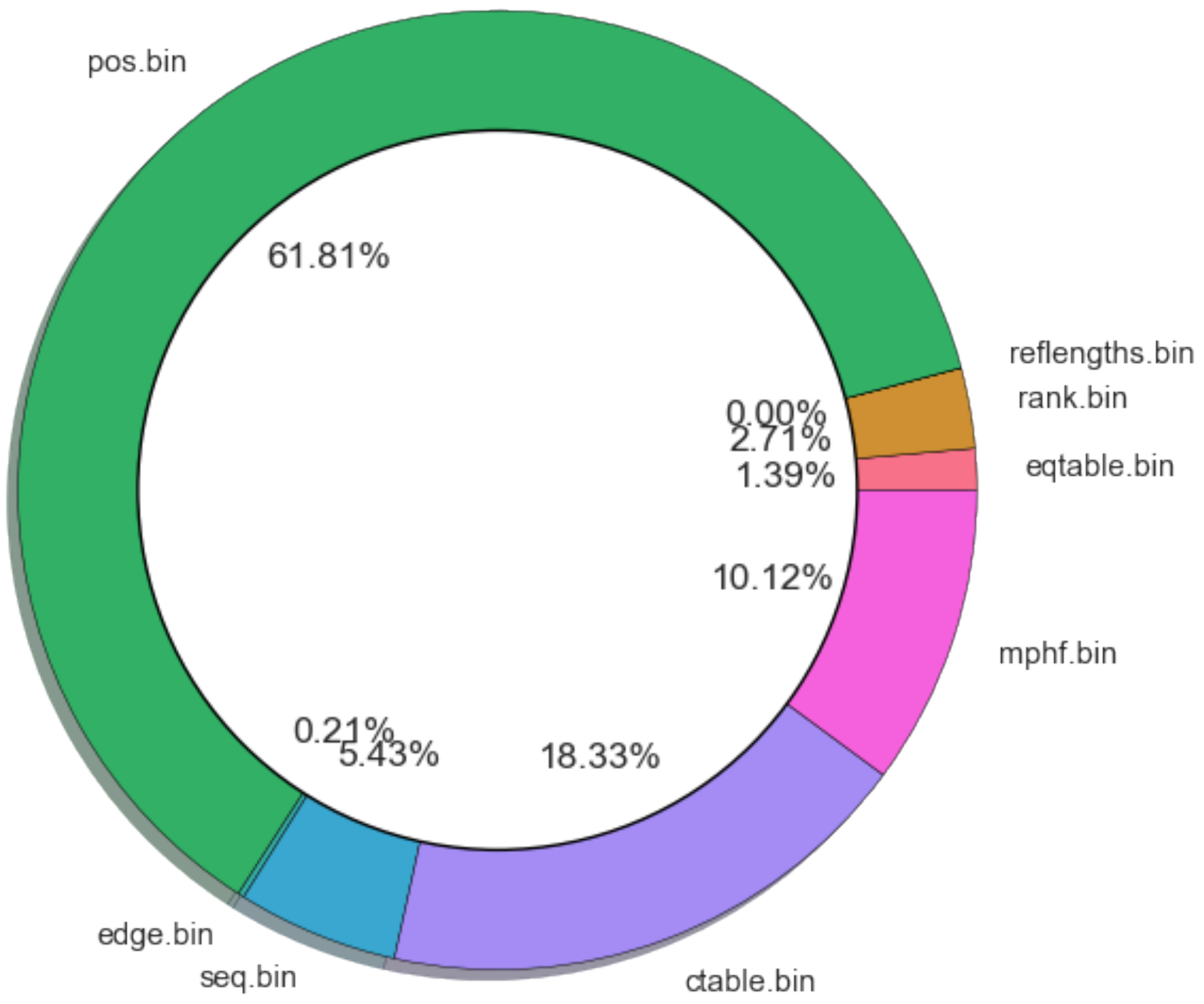
Optionally: explicit edge table, equivalence class table

The **dense** Pufferfish index

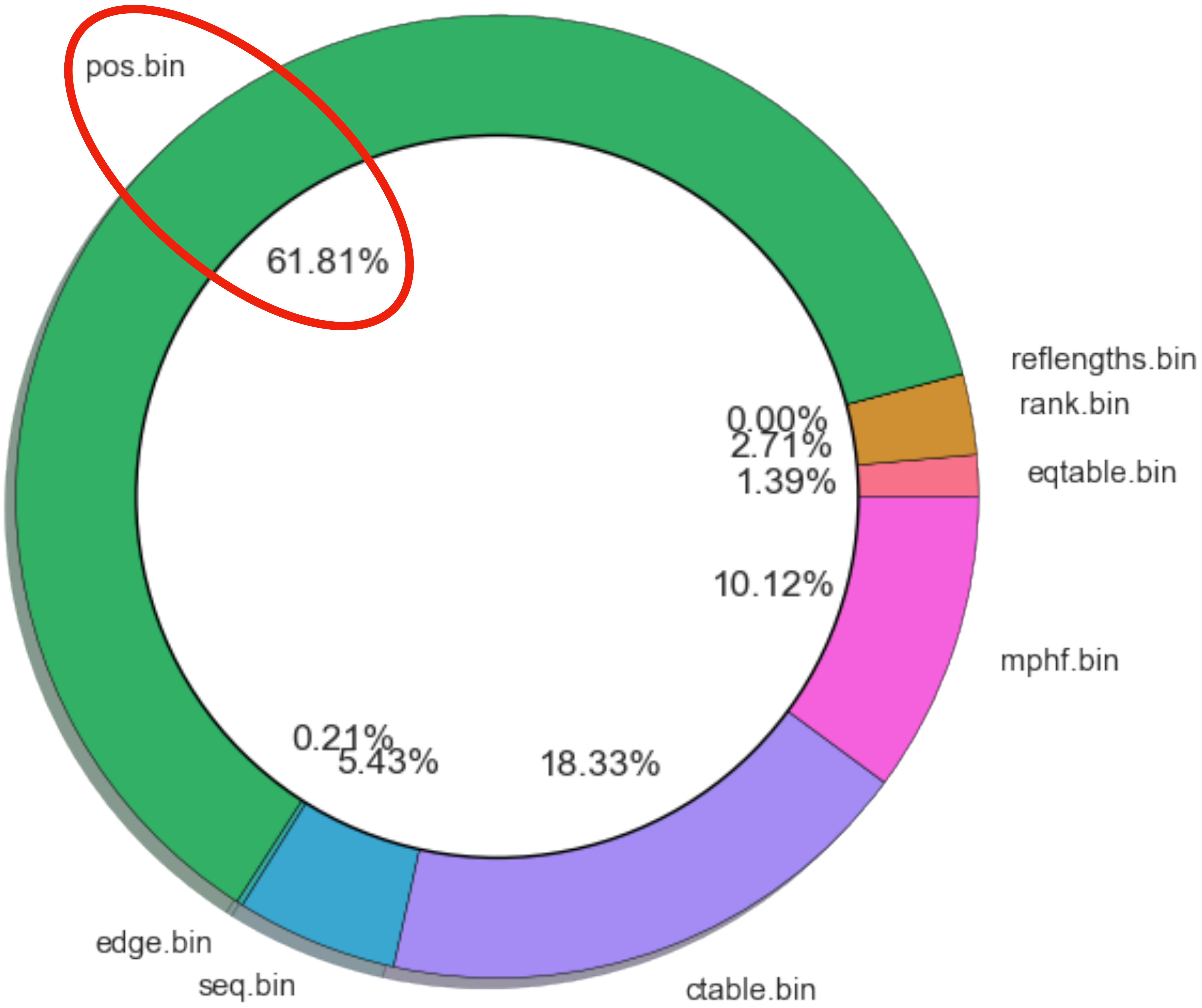


Optionally: explicit edge table, equivalence class table

Who's the culprit?

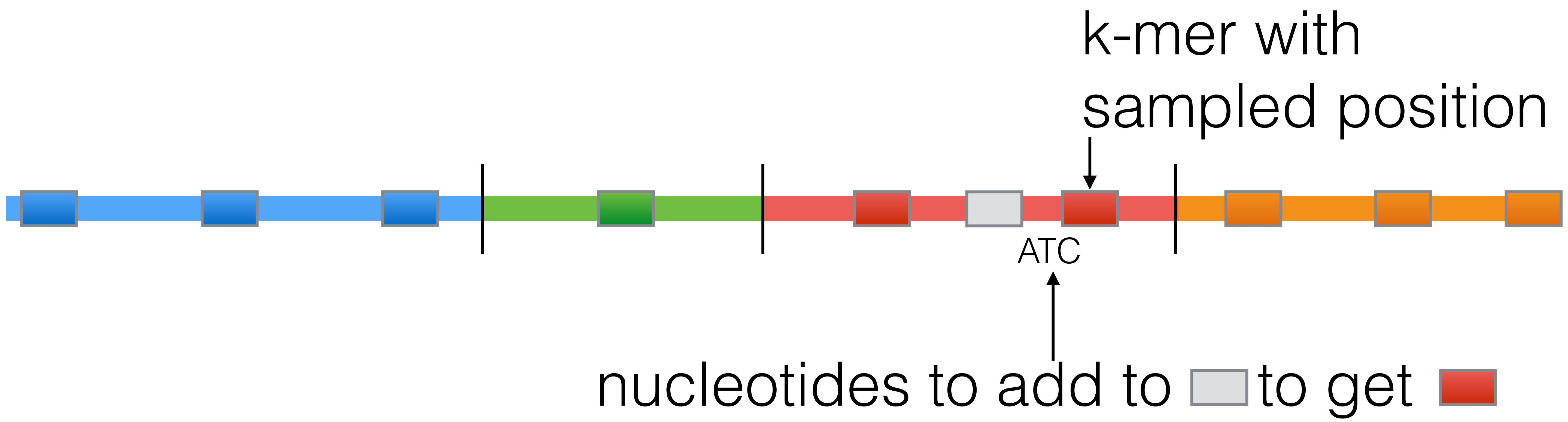


Who's the culprit?



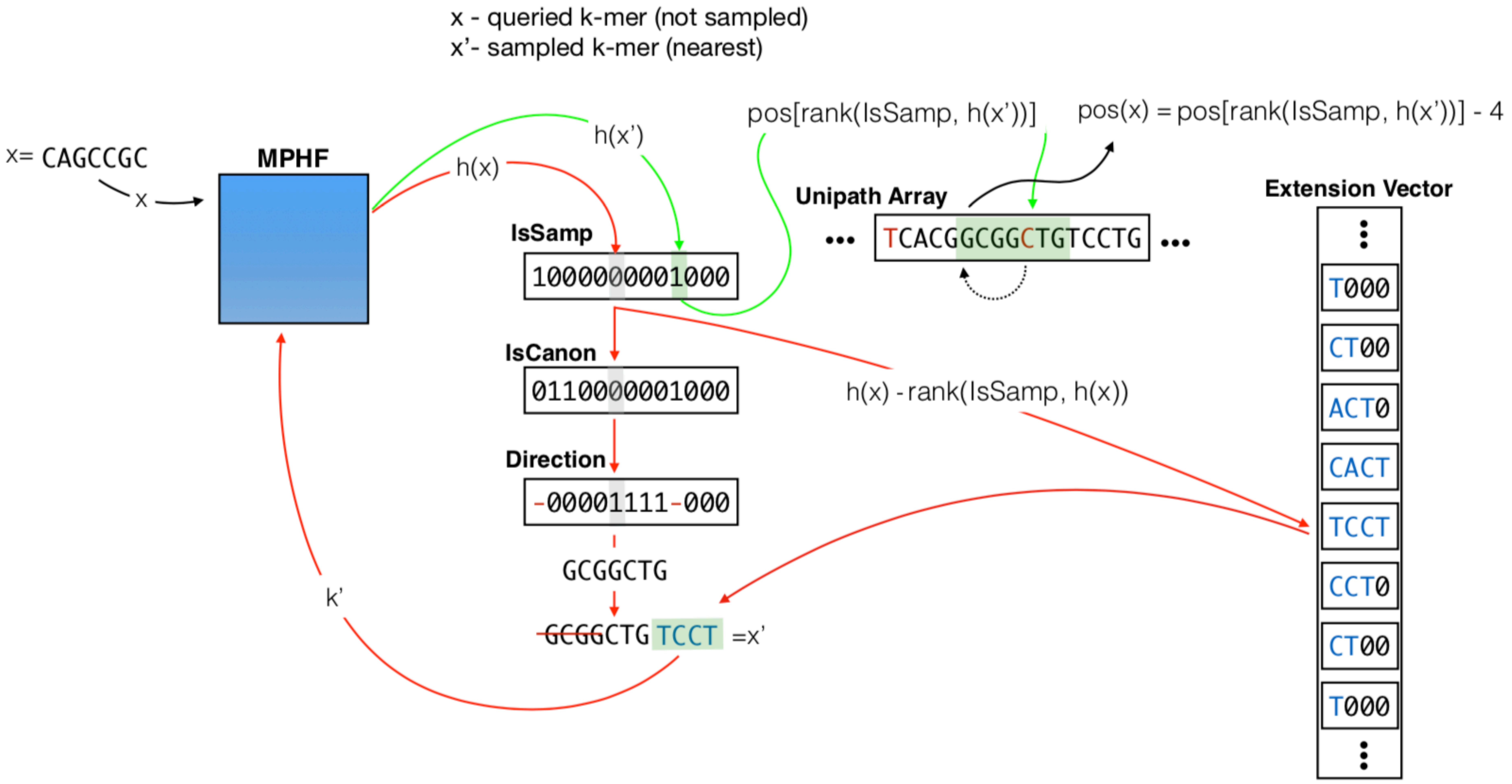
The **sparse** Pufferfish index

In large indices, the position table *dominates* index size



Intuition: Successors and predecessors in unipaths are *globally unique*, instead of storing position information for all k-mers, store positions only at sampled "landmarks" and say how to navigate to these landmarks (similar to bi-directional sampling in the FM-index).

The **sparse** Pufferfish index (in detail)

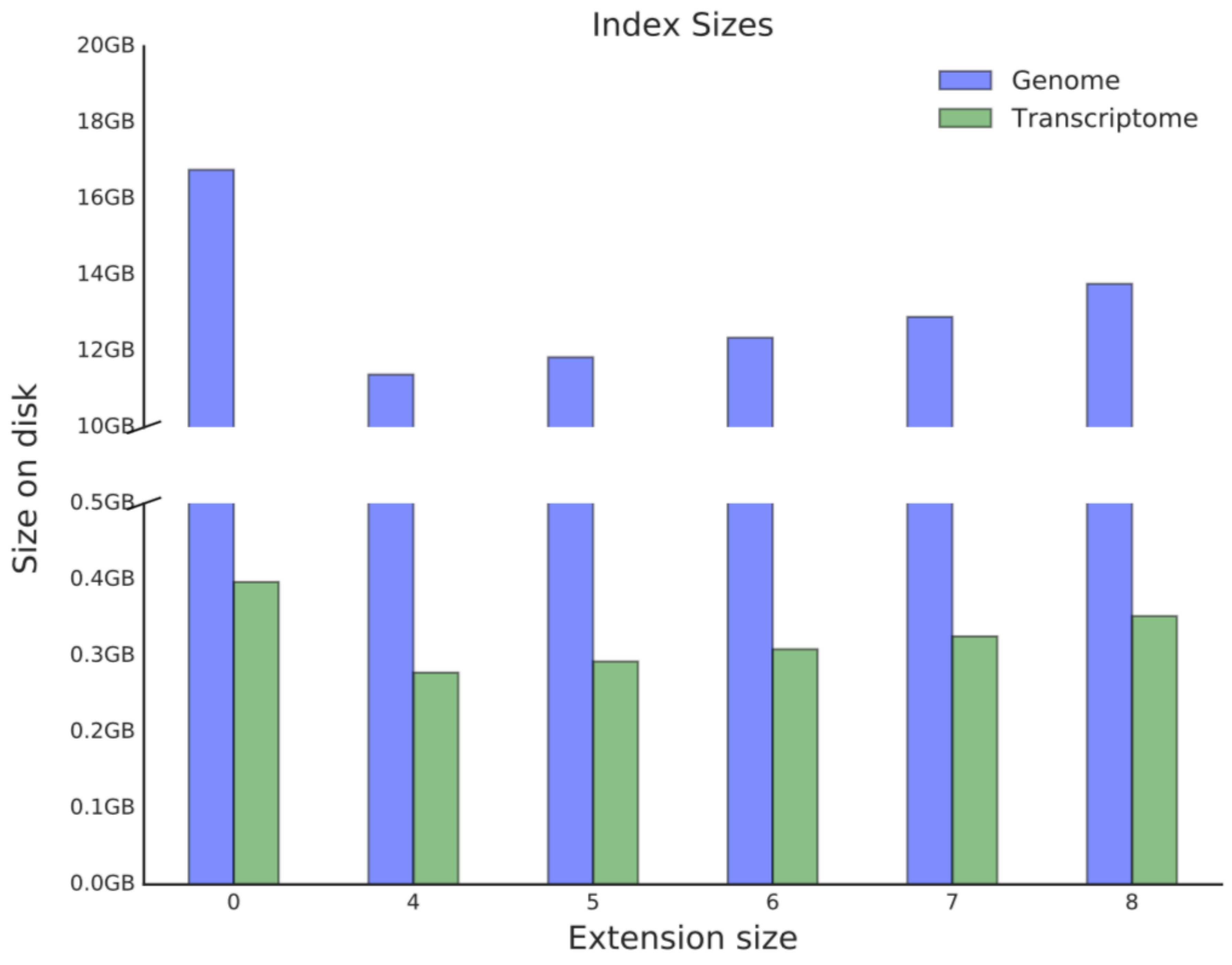


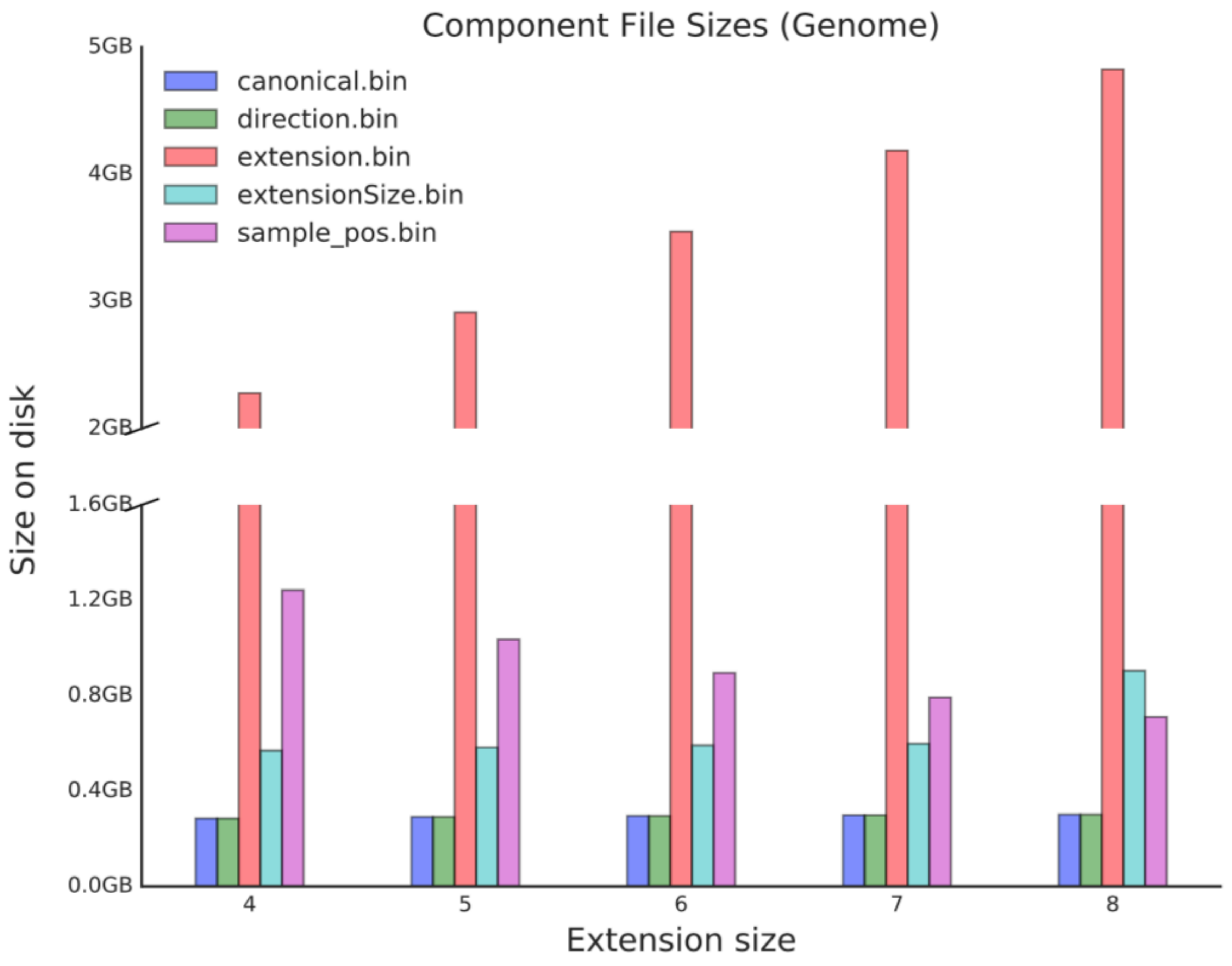
What sampling factor is right?

Tradeoff : Sparser sampling → less space but slower lookup

Fastest : Sampling factor s > 2·e+1 (Still a range of sizes)

Smallest : Extension size = 1, sampling = s





Index space & K-mer query time

Space of index + query in RAM

Tool	Memory (MB)		
	Human Transcriptome	Human Genome	Bacterial Genome
BWA	308	4,439	27,535
kallisto	3,336	110,464	232,353
pufferfish dense	454	17,684	41,532
pufferfish sparse	341	12,533	30,565

#Li, H. (2013). Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arXiv Preprint arXiv:1303.3997.

^Bray, N. L., Pimentel, H., Melsted, P., and Pachter, L. (2016). Near-optimal probabilistic RNA-seq quantification. Nature Biotechnology, 34(5), 525–527.

Index space & K-mer query time

Time to look up all fixed-length substrings in an experiment

Tool	Time (h:m:s)		
	Human Transcriptome	Human Genome	Bacterial Genome
BWA	0:17:35	0:50:31	0:14:05
kallisto	0:02:01	0:19:11	0:22:25
pufferfish dense	0:02:46	0:10:37	0:06:03
pufferfish sparse	0:08:34	0:22:11	0:08:26
# queries:	747,842,900	7,508,576,020	509,143,360

Pufferfish summary (part 1)

- To keep memory usage reasonable, we have to be quite careful about our hashing-based schemes.

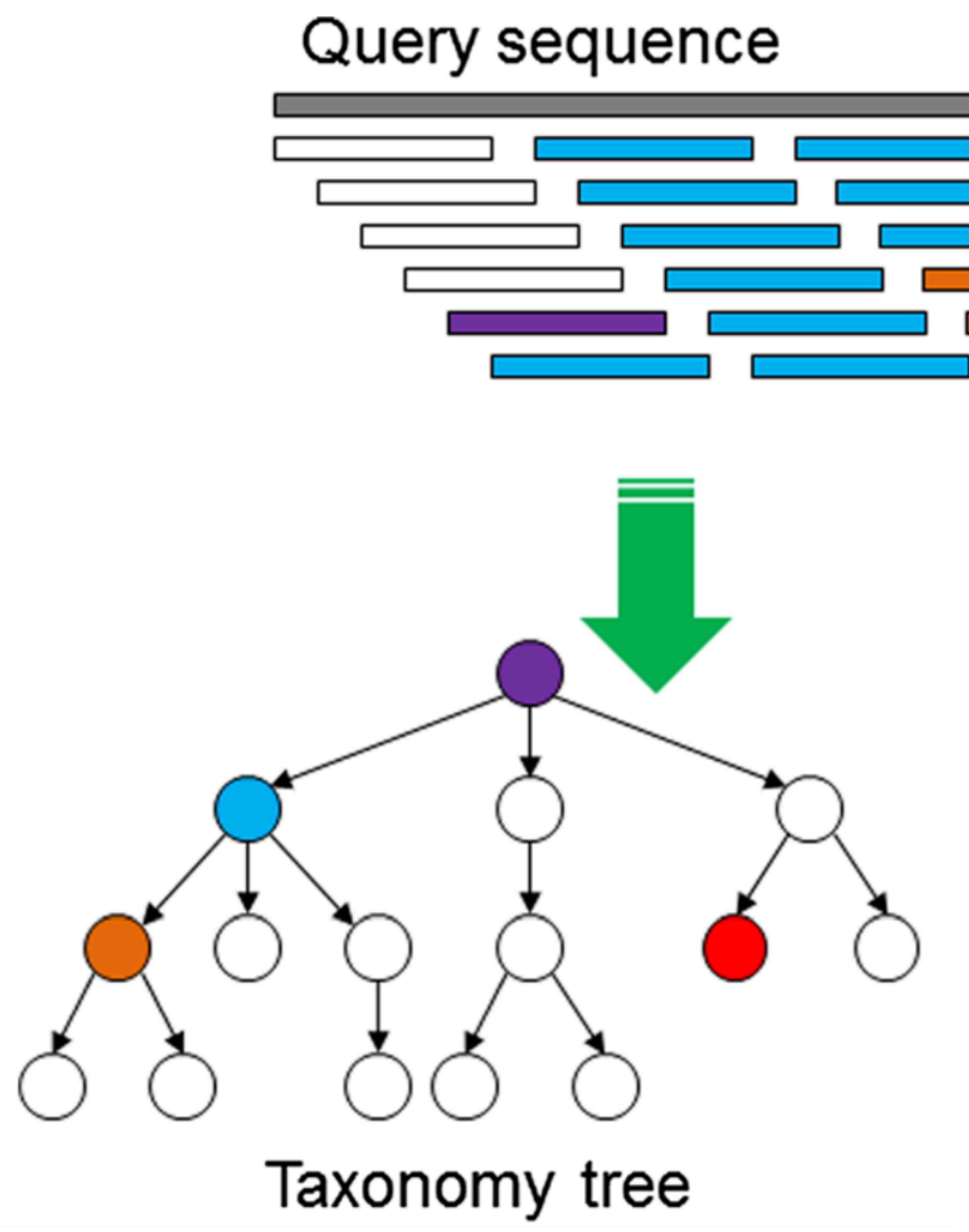
- The dense pufferfish index strikes a good balance between index space and raw query speed.

- At a constant factor (though not asymptotic) cost, index size is tunable with our sampling scheme.

- At least for fixed-length patterns, a good hashing approach can be *much faster* than (still asymptotically-optimal) full-text indexes.

An example application of Pufferfish

- Taxonomic read classification — for each read, assign it to the taxon (strain, species, genus) from which we think it derived. Related to, ***but distinct from***, taxonomic abundance estimation.



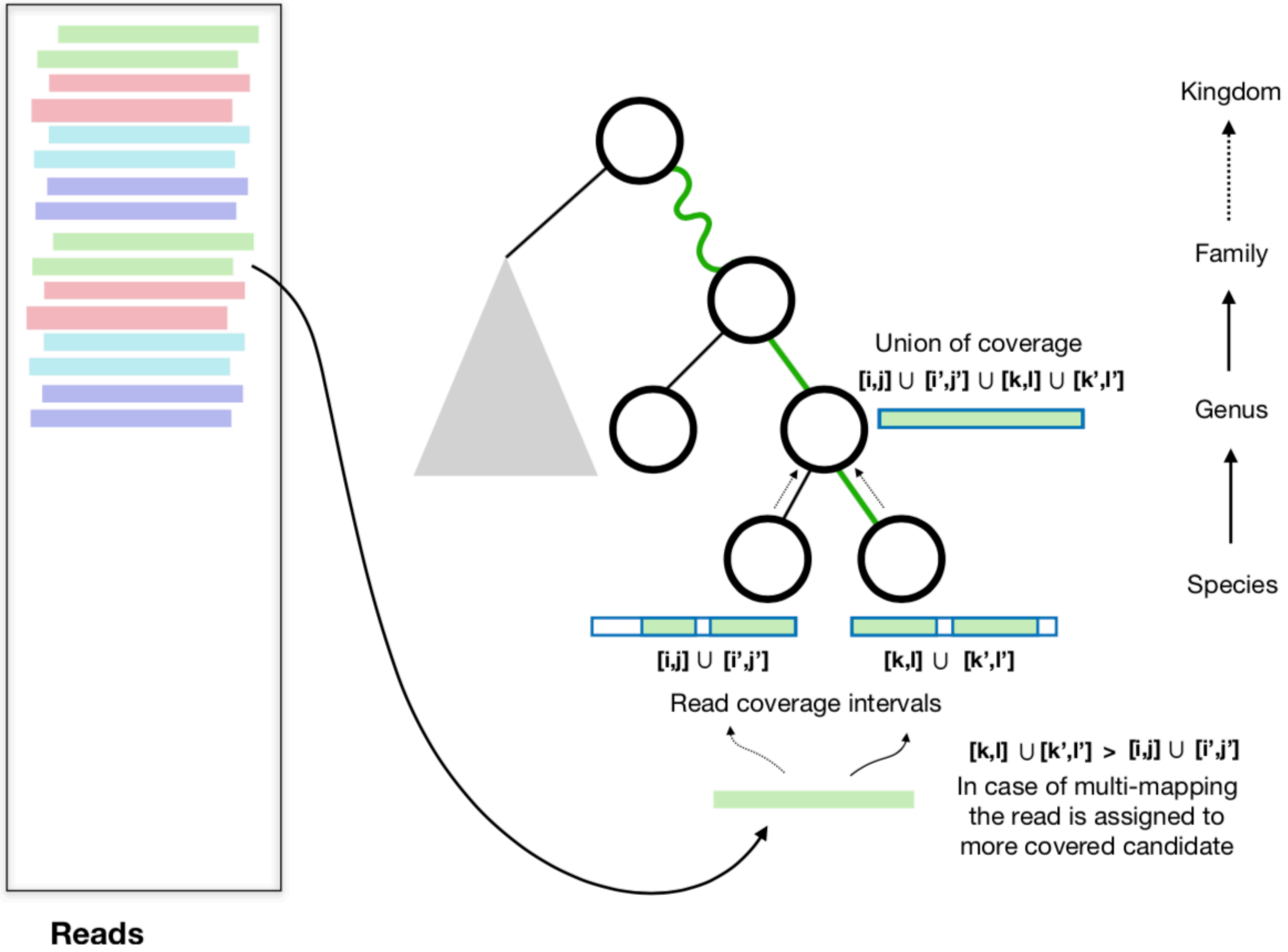
Figures adapted from: Wood, D.E. and Salzberg, S.L., 2014. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, *15*(3), p.R46.

Pufferfish taxonomic assignment

We adopt what is essentially the algorithm of *Kraken**, but replace k-mer counting with lightweight mapping.

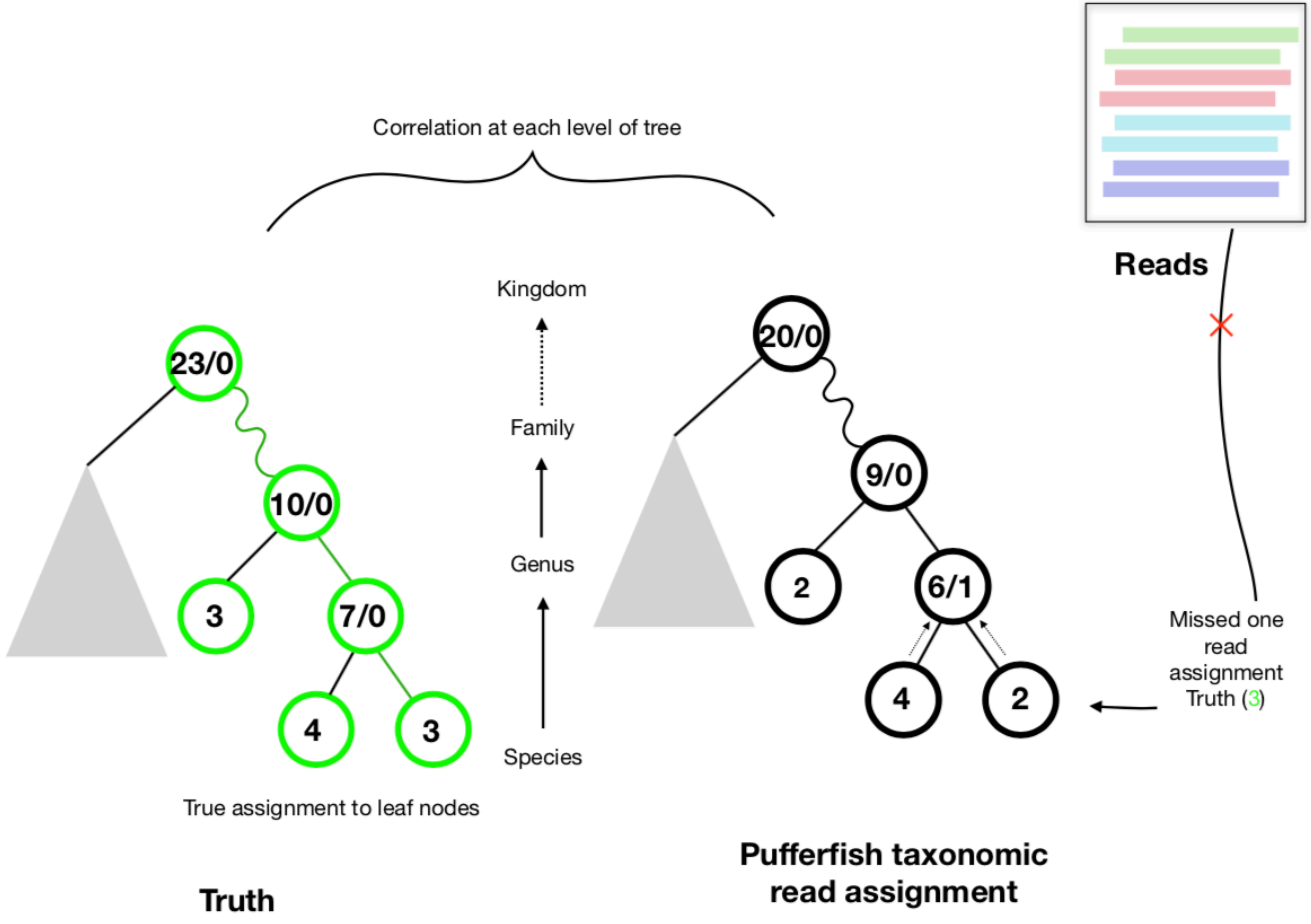
This *enforces positional & orientation consistency* of matches

- Score all root-to-leaf (RTL) paths
- Assign read to leaf of highest-scoring path
- In case of tie, assign read to LCA of all highest-scoring paths.

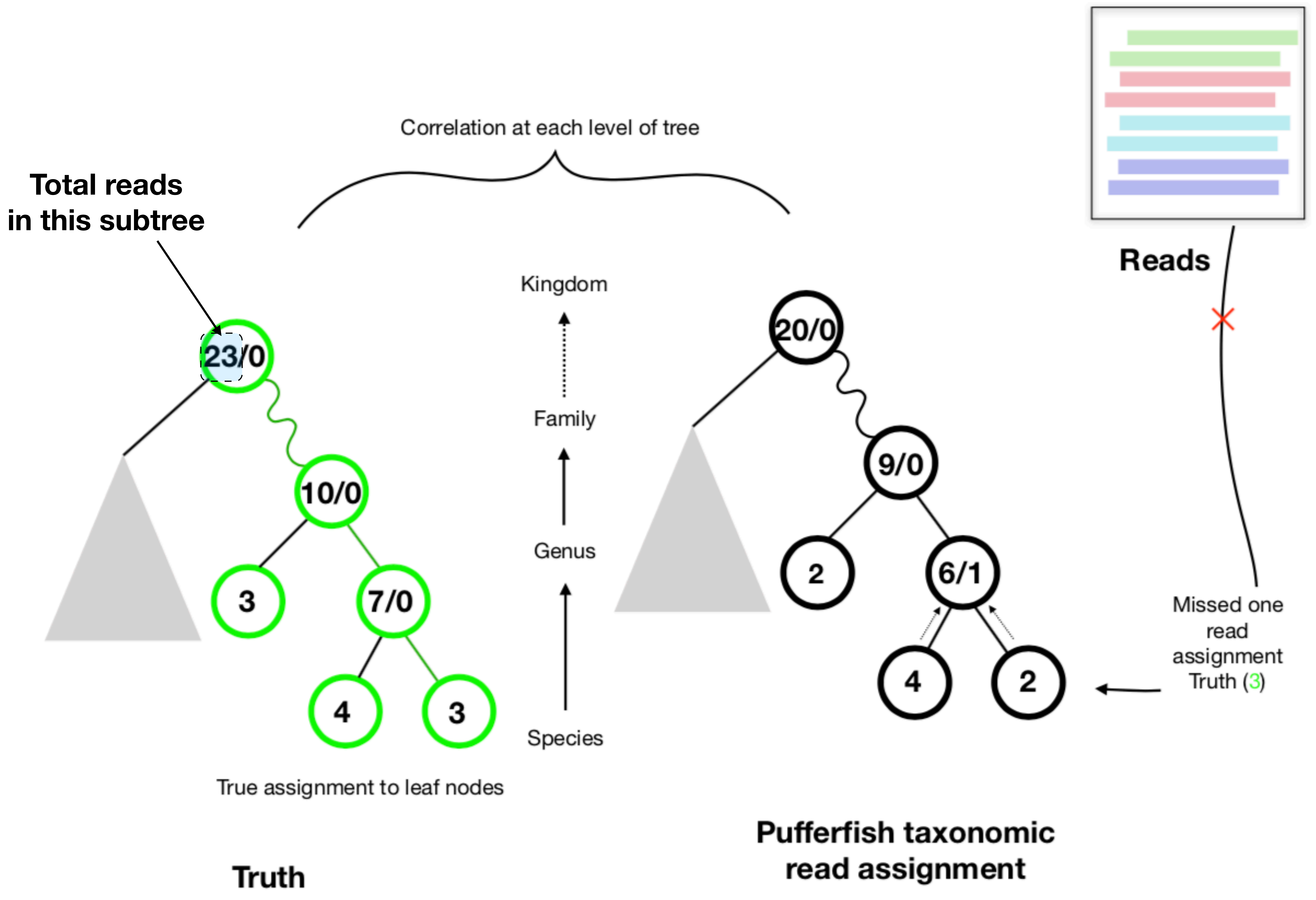


*Wood, D.E. and Salzberg, S.L., 2014. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, *15*(3), p.R46.

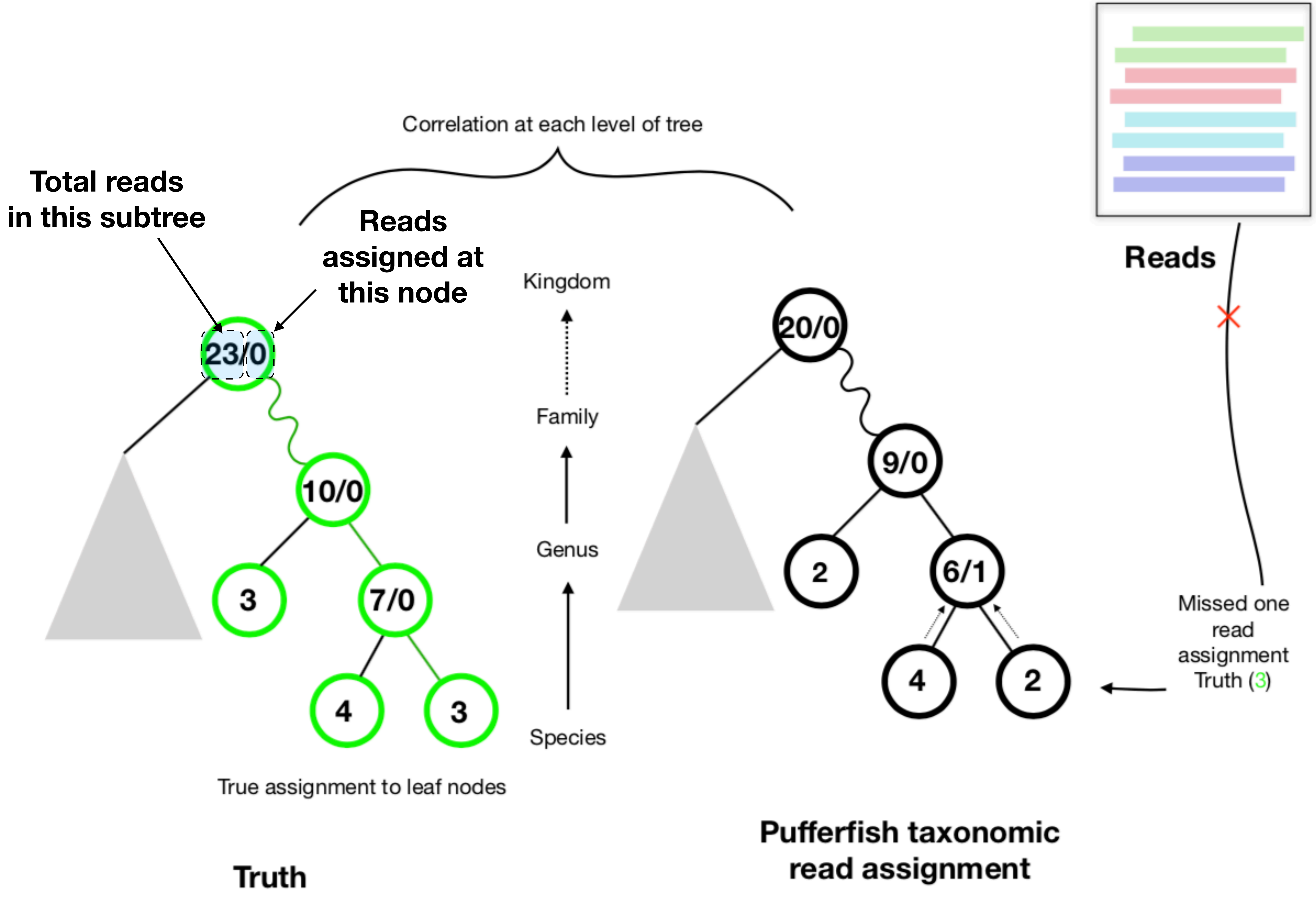
“Whole taxonomy” accuracy assessment



“Whole taxonomy” accuracy assessment

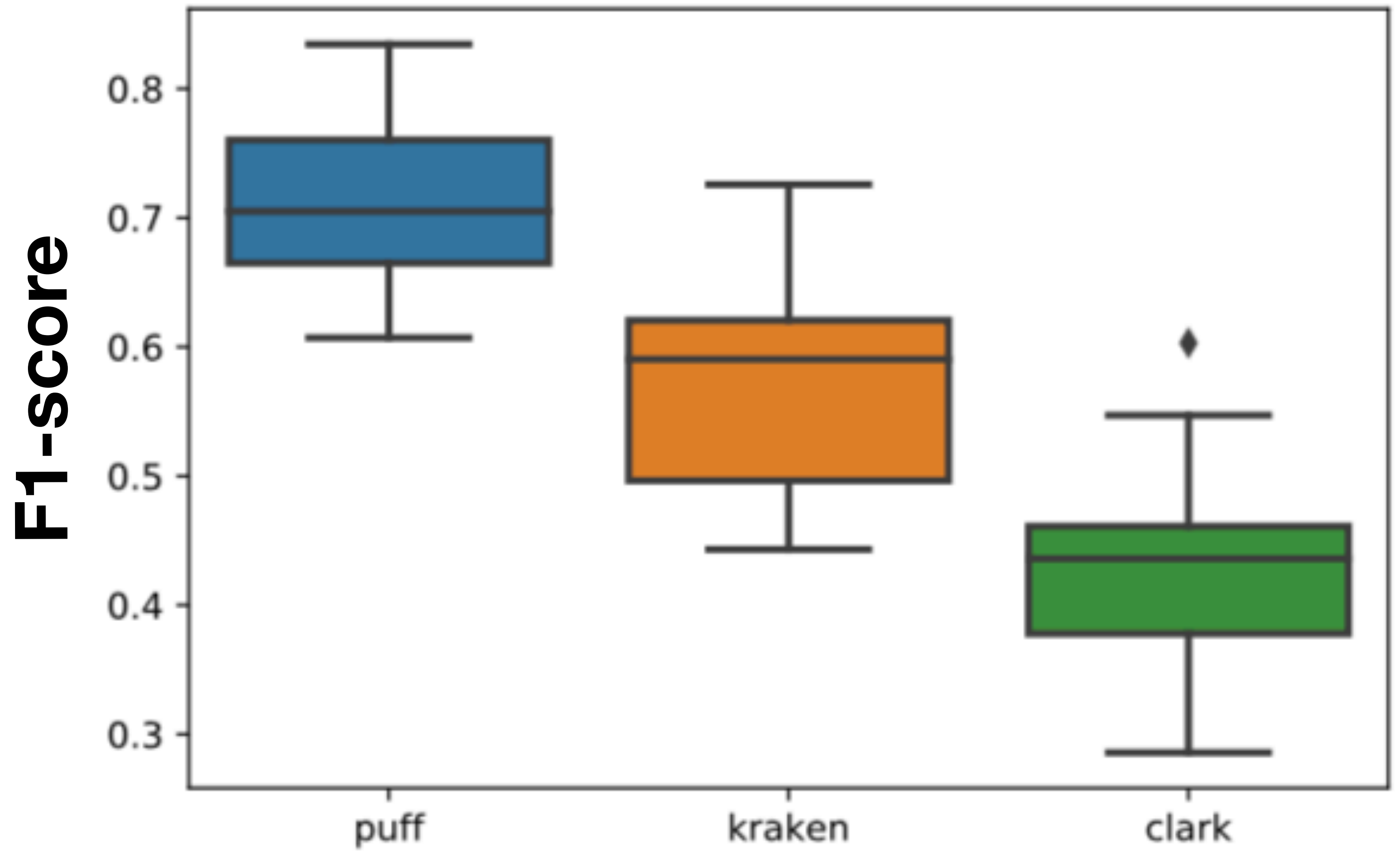


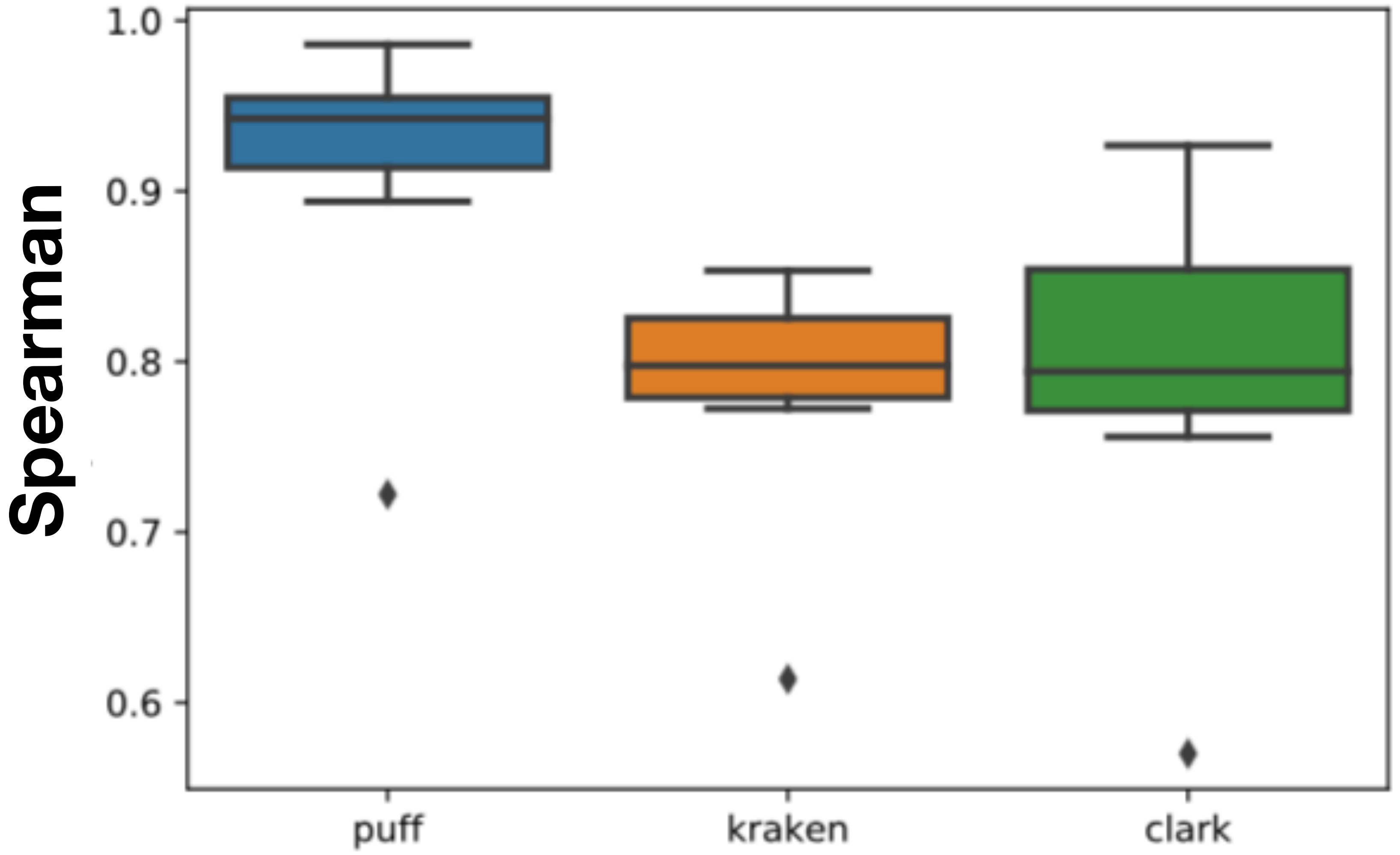
“Whole taxonomy” accuracy assessment



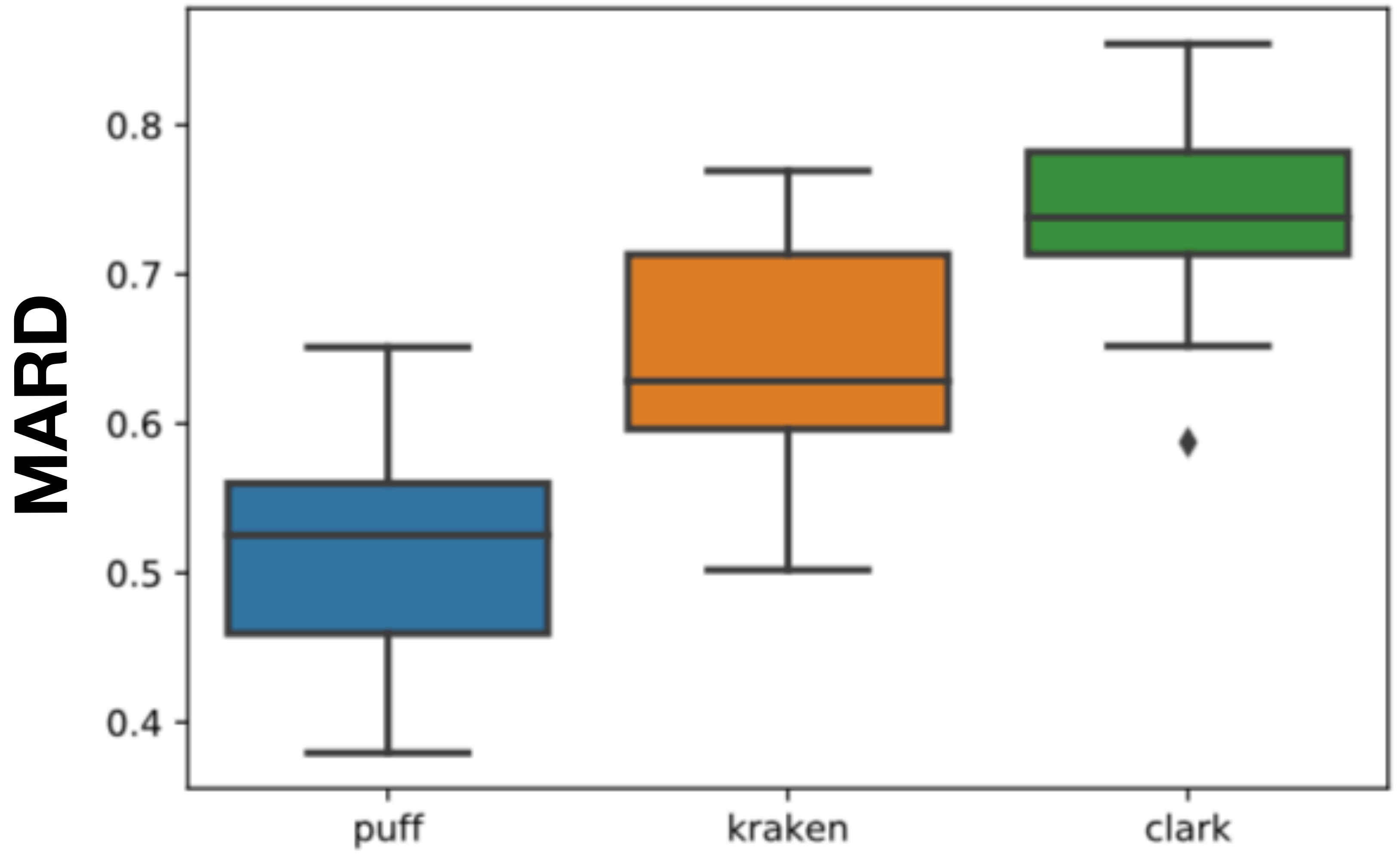
Pufferfish taxonomic assignment

Higher is better





Lower is better (distance)



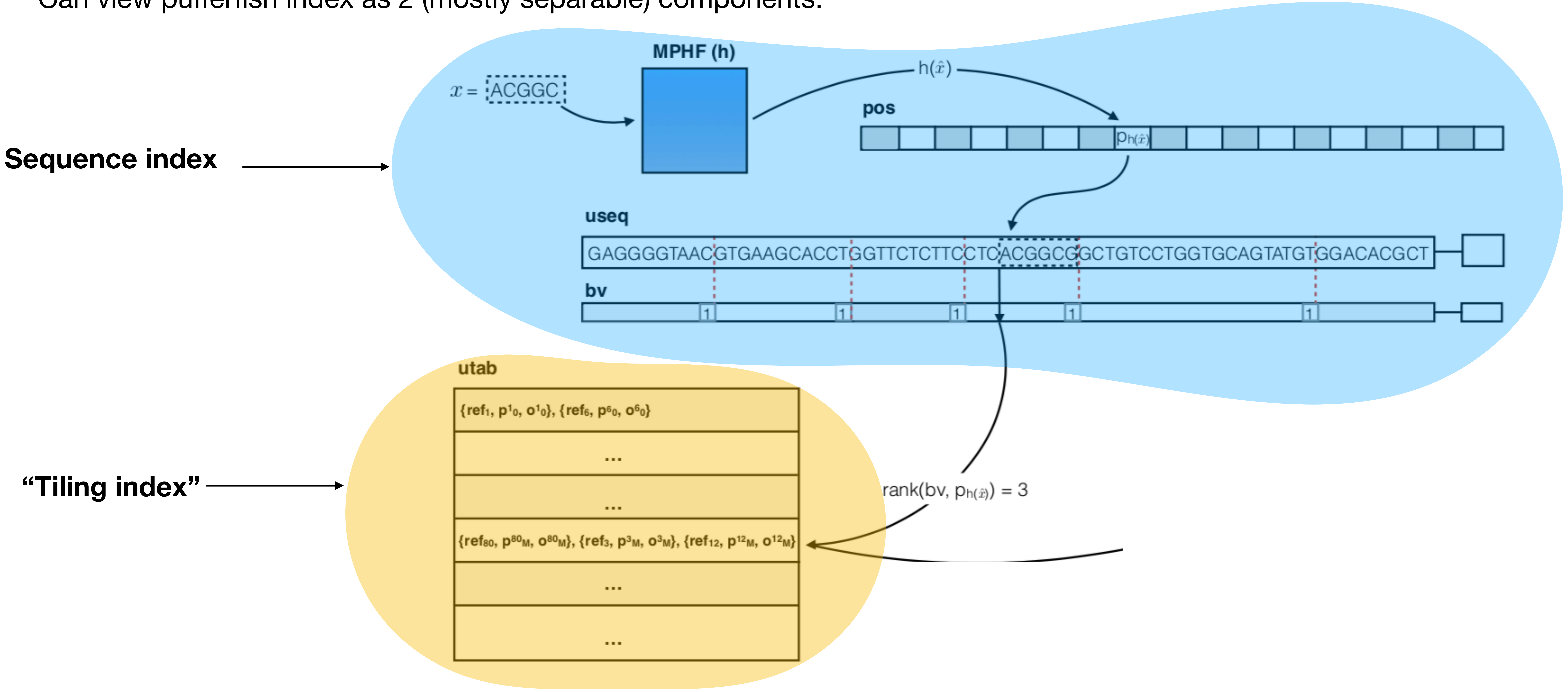
Simulated data from : McIntyre, et al. (2017).
Comprehensive benchmarking and ensemble approaches for metagenomic classifiers. Genome Biology, 18(1).

Simulations:
(LC1-8, HC1, HC2)

Doing even better for the sequence table

Pufferfish was introduced in 2017 and published in 2018. The field has come a long way since then; particularly in terms of better representations of the sequence part of the index.

Can view pufferfish index as 2 (mostly separable) components:



Doing even better for the sequence table

More recent improvements to the sequence index component:

BLight: efficient exact associative structure for k-mers

Camille Marchet, Mael Kerbiriou, Antoine Limasset

Bioinformatics, Volume 37, Issue 18, 15 September 2021, Pages 2858–2865,
https://doi.org/10.1093/bioinformatics/btab217
Published: 03 April 2021 **Article history** ▾

JOURNAL ARTICLE

Sparse and skew hashing of K-mers

Giulio Ermanno Pibiri

Bioinformatics, Volume 38, Issue Supplement_1, July 2022, Pages i185–i194,
https://doi.org/10.1093/bioinformatics/btac245
Published: 27 June 2022

Sparse and Skew Hashing of K-Mers represents the current state-of-the-art and builds on both pufferfish and BLight.

Both pufferfish and BLight take advantage of the idea of minimizers.

SSHash

JOURNAL ARTICLE

Sparse and skew hashing of K-mers

Giulio Ermanno Pibiri

Bioinformatics, Volume 38, Issue Supplement_1, July 2022, Pages i185–i194,
https://doi.org/10.1093/bioinformatics/btac245
Published: 27 June 2022

Following slides adapted from a presentation created by Jason Fan

Motivation

1. Associative data-structures, or *dictionaries*, that map k-mers key to sequence analysis.

Motivation

1. Associative data-structures, or *dictionaries*, that map k-mers key to sequence analysis.
2. Goal is to support fast queries and space efficient representations of: (k-mer, value) pairs in the general case.

Motivation

1. Associative data-structures, or *dictionaries*, that map k-mers key to sequence analysis.
2. Goal is to support fast queries and space efficient representations of: (k-mer, value) pairs in the general case.
3. Many groups have been thinking about efficient ways to build and store these data structures.

Motivation

1. Associative data-structures, or *dictionaries*, that map k-mers key to sequence analysis.
2. Goal is to support fast queries and space efficient representations of: (k-mer, value) pairs in the general case.
3. Many groups have been thinking about efficient ways to build and store these data structures.

In this paper, given a k-mer set S of size n. We want data structure that supports:

Motivation

1. Associative data-structures, or *dictionaries*, that map k-mers key to sequence analysis.
2. Goal is to support fast queries and space efficient representations of: (k-mer, value) pairs in the general case.
3. Many groups have been thinking about efficient ways to build and store these data structures.

In this paper, given a k-mer set S of size n. We want data structure that supports:

1. Lookup(g) that uniquely maps any $g \in S$ to an integer $0 \leq i < n$

Motivation

1. Associative data-structures, or *dictionaries*, that map k-mers key to sequence analysis.
2. Goal is to support fast queries and space efficient representations of: (k-mer, value) pairs in the general case.
3. Many groups have been thinking about efficient ways to build and store these data structures.

In this paper, given a k-mer set S of size n. We want data structure that supports:

1. Lookup(g) that uniquely maps any $g \in S$ to an integer $0 \leq i < n$
2. Access (i) that returns a k-mer g s.t Lookup(g) = i.

Motivation

1. Associative data-structures, or *dictionaries*, that map k-mers key to sequence analysis.
2. Goal is to support fast queries and space efficient representations of: (k-mer, value) pairs in the general case.
3. Many groups have been thinking about efficient ways to build and store these data structures.

In this paper, given a k-mer set S of size n. We want data structure that supports:

1. Lookup(g) that uniquely maps any $g \in S$ to an integer $0 \leq i < n$
2. Access (i) that returns a k-mer g s.t Lookup(g) = i.

Note: Access (i) is really only easy in this paper since values are indices.

Key idea: “streaming” queries

Many applications care about querying adjacent k-mers on a string.
Where consecutive k-mers on a string are queried.

Key idea: “streaming” queries

Many applications care about querying adjacent k-mers on a string. Where consecutive k-mers on a string are queried.

Some data-structures are optimized to handle this. One example is pufferfish…

Key idea: “streaming” queries

Many applications care about querying adjacent k-mers on a string. Where consecutive k-mers on a string are queried.

Some data-structures are optimized to handle this. One example is pufferfish…

…which implements a cache that exploits the fact that consecutive k-mers likely land in the same contig (in the same set of references).

Key idea: “streaming” queries

Many applications care about querying adjacent k-mers on a string. Where consecutive k-mers on a string are queried.

Some data-structures are optimized to handle this. One example is pufferfish…

…which implements a cache that exploits the fact that consecutive k-mers likely land in the same contig (in the same set of references).

Minimizers can also be exploited

Minimizers: Sparsifying k-mers

The **minimizer** of a k-mer is the smallest length m sub-sequence of the k-mer under some ordering σ

ACTGACCCGTAGC **k-mer X (k=13)**

ACTGACCCGTAGC **minimizer of x (for m=3,** σ = alphabetical ordering**)**

This can be useful for partitioning / grouping k-mers

ACTGACCCGTAGCGCTAGATAAC

ACTGACCCGTAGCGCTAGATAAC

ACTGACCCGTAGCGCTAGATAAC

All k-mers in this window of length 19 share the *same* minimizer; they are called a **super k-mer**

A super k-mer can have length between k and 2k-m; provides a way to group k-mers looking only at it's actual sequence!

> Bioinformatics. 2004 Dec 12;20(18):3363-9. doi: 10.1093/bioinformatics/bth408. Epub 2004 Jul 15.

Reducing storage requirements for biological sequence comparison

Michael Roberts [1], Wayne Hayes, Brian R Hunt, Stephen M Mount, James A Yorke

ARTICLE

Winnowing: local algorithms for document fingerprinting

Authors: Saul Schleimer, Daniel S. Wilkerson, Alex Aiken Authors Info & Claims

SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data • June 2003 • Pages 76–85 • https://doi.org/10.1145/872757.872770

SSHash

SSHash is much like pufferfish but with a few important optimizations:

1. Instead of sampling positions with a constant stride length… sample based on minimizers and store the positions of all super k-mers containing these minimizers.
2. At query time, given a k-mer g. Find its minimizer r, lookup all occurrences of r, and return the Lookup(g) as appropriate

So how to sample based on minimizers?

Super k-mers := the maximal set of consecutive k-mers on a reference sequence that share the same minimizer (sequence).

So how to sample based on minimizers?

Super k-mers := the maximal set of consecutive k-mers on a reference sequence that share the same minimizer (sequence).

AAGCAACTGGT

AAGCAACTGGT

AAGCAACTGGT

So how to sample based on minimizers?

Super k-mers := the maximal set of consecutive k-mers on a reference sequence that share the same minimizer (sequence).

AAGCAACTGGT

AAGCAACTGGT

AAGCAACTGGT

This yields a **"bucketed" partitioning** of the reference where a bucket B_r contains all the super k-mers on the reference with minimizer r.

So how to sample based on minimizers?

Super k-mers := the maximal set of consecutive k-mers on a reference sequence that share the same minimizer (sequence).

AAGCAACTGGT

AAGCAACTGGT

AAGCAACTGGT

This yields a **"bucketed" partitioning** of the reference where a bucket B_r contains all the super k-mers on the reference with minimizer r.

The intuition is that B_r is usually small, and that you can exhaustively search for matches to a query k-mer with minimizer r in B_r.

So how to sample based on minimizers?

Given p strings (unitigs), S, with total length N

1. ***useq*** *:= the sequence of unitigs*

So how to sample based on minimizers?

Given p strings (unitigs), S, with total length N

1. ***useq*** *:= the sequence of unitigs*
2. ***endpoints***, such that ***useq***[***endpoints***[i]] is the last base of a unitig in useq.

So how to sample based on minimizers?

Given p strings (unitigs), S, with total length N

1. ***useq*** *:= the sequence of unitigs*
2. ***endpoints***, such that ***useq***[***endpoints***[i]] is the last base of a unitig in useq.
3. f: a MPHF over the set of minimizers of length m on S.

So how to sample based on minimizers?

Given p strings (unitigs), S, with total length N

1. ***useq*** *:= the sequence of unitigs*
2. ***endpoints***, such that ***useq***[***endpoints***[i]] is the last base of a unitig in useq.
3. f: a MPHF over the set of minimizers of length m on S.
4. ***sizes***, such that ***sizes***[i + 1] – ***sizes***[i] = $\left|B_r\right|$ when $f(r)$= i.

So how to sample based on minimizers?

Given p strings (unitigs), S, with total length N

1. ***useq*** *:= the sequence of unitigs*
2. ***endpoints***, such that ***useq***[***endpoints***[i]] is the last base of a unitig in useq.
3. f: a MPHF over the set of minimizers of length m on S.
4. ***sizes***, such that ***sizes***[i + 1] – ***sizes***[i] = $\left|B_r\right|$ when$f(r)$= i.
5. ***offsets***, such that for a minimizer r, with ***sizes***[$f(r)$] = ***begin***, ***offsets***[***begin***, ***begin*** + $\left|B_r\right|$] contain the absolute positions of each super k-mer with minimizer r on ***useq***.

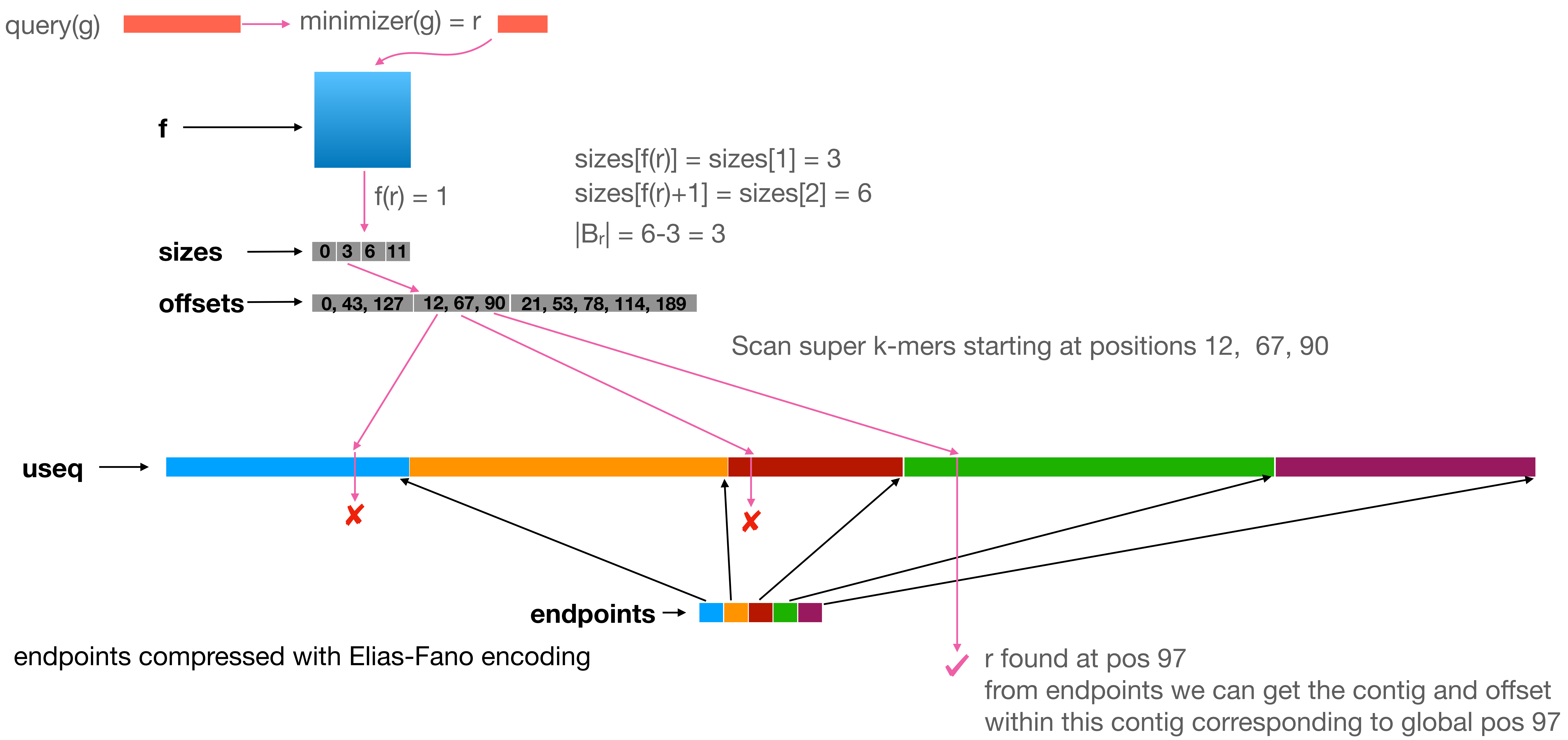
SSHash

SSHash

SSHash is just like pufferfish.

1. Instead of sampling positions with a constant stride length… sample based on minimizers and store the positions of super-kmers containing these minimizers
2. At query time, given a k-mer g. Find its minimizer r, lookup all occurrences of r, and return the Lookup(g) as appropriate

SShash (without the skew) visually



Query

Given a k-mer g:

1. $r = \text{minimizer}_m(g)$

Query

Given a k-mer g:

1. $r = \mathrm{minimizer}_m(g)$
2. begin $=$ ***sizes***$[f(r)]$, end $=$ **sizes**$[f(r)$ + 1]

Query

Given a k-mer g:

1. $r = \text{minimizer}_m(g)$
2. begin $=$ ***sizes***[$f(r)$], end $=$ **sizes**[$f(r)$ + 1]
3. Check that k-mer at ***useq***[***offsets***[begin]] has minimizer r

Query

Given a k-mer g:

1. $r = \mathrm{minimizer}_m(g)$
2. begin $=$ ***sizes***[$f(r)$], end $=$ **sizes**[$f(r)$ + 1]
3. Check that k-mer at ***useq***[***offsets***[begin]] has minimizer r
4. For each t in ***offsets***[begin, end) "*scan the super-kmer at position t on useq*".

Query

Given a k-mer g:

1. $r = \text{minimizer}_m(g)$
2. begin $=$ ***sizes***$[f(r)]$, end $=$ **sizes**$[f(r)$ + 1]
3. Check that k-mer at ***useq***[***offsets***[begin]] has minimizer r
4. For each t in ***offsets***[begin, end) "*scan the super-kmer at position t on useq*".
   a. Let t_{end} be smallest entry in endpoints greater than t.
   b. Let $l = \min(2k - m, t_{\text{end}} - t)$

Query

Given a k-mer g:

1. $r = \text{minimizer}_m(g)$
2. begin $=$ ***sizes***$[f(r)]$, end $=$ **sizes**$[f(r)$ + 1]
3. Check that k-mer at ***useq***[***offsets***[begin]] has minimizer r
4. For each t in ***offsets***[begin, end) "*scan the super-kmer at position t on useq*".
    a. Let t_{end} be smallest entry in endpoints greater than t.
    b. Let $l = \min(2k - m, t_{\text{end}} - t)$
    c. Scan string ***useq***[$t, t + l$] for exact match with g.

Query

Given a k-mer g:

1. $r = \text{minimizer}_m(g)$
2. begin $=$ ***sizes***$[f(r)]$, end $=$ **sizes**$[f(r)$ + 1]
3. Check that k-mer at ***useq***[***offsets***[begin]] has minimizer r
4. For each t in ***offsets***[begin, end) "*scan the super-kmer at position t on useq*".
    a. Let t_{end} be smallest entry in endpoints greater than t.
    b. Let $l = \min(2k - m, t_{\text{end}} - t)$
    c. Scan string ***useq***[$t, t + l$] for exact match with g.
    d. If a match is found at position w on ***useq***[$t, t + l$], return $w + t - j(k - 1)$

Query

Given a k-mer g:

1. $r = \text{minimizer}_m(g)$
2. begin $=$ ***sizes***$[f(r)]$, end $=$ **sizes**$[f(r)$ + 1]
3. Check that k-mer at ***useq***[***offsets***[begin]] has minimizer r
4. For each t in ***offsets***[begin, end) "*scan the super-kmer at position t on useq*".
   a. Let t_{end} be smallest entry in endpoints greater than t.
   b. Let $l = \min(2k - m, t_{\text{end}} - t)$
   c. Scan string ***useq***[$t, t + l$] for exact match with g.
   d. If a match is found at position w on ***useq***[$t, t + l$], return $w + t - j(k - 1)$
      a. Where j is the number of unitigs encoded on ***useq*** before position t.

A note on super k-mer lengths

Super k-mers := the maximal set of consecutive k-mers on a reference sequence that share the same minimizer (sequence).

AAGCAACTGGT

AAGCAACTGGT

AAGCAACTGGT

A note on super k-mer lengths

Super k-mers := the maximal set of consecutive k-mers on a reference sequence that share the same minimizer (sequence).

AAGCAACTGGT

AAGCAACTGGT

AAGCAACTGGT

Super k-mers have length "at most 2k – m"...

A note on super k-mer lengths

Super k-mers := the maximal set of consecutive k-mers on a reference sequence that share the same minimizer (sequence).

AAGCAACTGGT
AAGCAACTGGT
AAGCAACTGGT

Super k-mers have length “at most 2k – m”...

But not really, since you can have:

AAGCAACTGAAC
AAGCAACTGAAC
AAGC**AACTGAA**C
AAGCA**ACTGAAC**

A note on super k-mer lengths

Super k-mers := the maximal set of consecutive k-mers on a reference sequence that share the same minimizer (sequence).

AAGCAACTGGT
AAGCAACTGGT
AAGCAACTGGT

Super k-mers have length "at most 2k – m"...

But not really, since you can have:

AAGCAACTGAAC
AAGCAACTGAAC
AAGC**AACTGAA**C
AAGCA**ACTGAAC**

The simple solution taken by SSHash is to simply truncate super-kmers of length greater than 2k-m into 2k-m blocks.

Skew hashing -- Bounding bucket sizes.

There are very few buckets that contain many super k-mers. But the size of these buckets may be large.

- e.g. largest bucket in human genome is ~36,000 super-kmers.

Skew hashing -- Bounding bucket sizes.

There are very few buckets that contain many super k-mers. But the size of these buckets may be large.

- e.g. largest bucket in human genome is ~36,000 super-kmers.

Note that though these buckets are “large” they are still small compared to the reference.

Skew hashing -- Bounding bucket sizes.

There are very few buckets that contain many super k-mers. But the size of these buckets may be large.

- e.g. largest bucket in human genome is ~36,000 super-kmers.

Note that though these buckets are “large” they are still small compared to the reference.

So not too many k-mers belong to these buckets.

Skew hashing -- Bounding bucket sizes.

There are very few buckets that contain many super k-mers. But the size of these buckets may be large.

e.g. largest bucket in human genome is ~36,000 super-kmers.

Note that though these buckets are "large" they are still small compared to the reference.

So not too many k-mers belong to these buckets.

Key idea: build a MPHFs over such k-mers directly to quickly associate them to the appropriate super k-mer, and its position in ***useq***.

Skew hashing

Given parameters ℓ, L, partition the buckets into L sets.

Let S_i be the set of k-mers belonging to any bucket B_r with:

$$2^i < \left|B_r\right| < 2^{i+1} \text{ for } \ell < i < L$$

$$2^L < \left|B_r\right| \text{ for } i = L$$

Skew hashing

Given parameters ℓ, L, partition the buckets into L sets.

Let S_i be the set of k-mers belonging to any bucket B_r with:

$$2^i < \left|B_r\right| < 2^{i+1} \text{ for } \ell < i < L$$

$$2^L < \left|B_r\right| \text{ for } i = L$$

Skew hashing

Given parameters ℓ, L, partition the buckets into L sets.

Let S_i be the set of k-mers belonging to any bucket B_r with:

$$2^i < \left|B_r\right| < 2^{i+1} \text{ for } \ell < i < L$$
$$2^L < \left|B_r\right| \text{ for } i = L$$

For each S_i, build an MPHF f_i.

And store compact vectors P_i, such that $P_i[f_i(g)] = q$, indicates that g occurs in the q-th super-kmer in some bucket B_r

Skew hashing

Given parameters ℓ, L, partition the buckets into L sets.

Let S_i be the set of k-mers belonging to any bucket B_r with:

$$2^i < \left|B_r\right| < 2^{i+1} \text{ for } \ell < i < L$$

$$2^L < \left|B_r\right| \text{ for } i = L$$

For each S_i, build an MPHF f_i.

And store compact vectors P_i, such that $P_i[f_i(g)] = q$, indicates that g occurs in the q-th super-kmer in some bucket B_r

Skew hashing

Given parameters ℓ, L, partition the buckets into L sets.

Let S_i be the set of k-mers belonging to any bucket B_r with:

$$2^i < \left|B_r\right| < 2^{i+1} \text{ for } \ell < i < L$$
$$2^L < \left|B_r\right| \text{ for } i = L$$

For each S_i, build an MPHF f_i.

And store compact vectors P_i, such that $P_i[f_i(g)] = q$, indicates that g occurs in the q-th super-kmer in some bucket B_r

NB: the sizing ensures optimal compacted P_i. Empirically, the compact vectors and MPHF are <1% of the SSHash size, and represent <2% of total k-mers

Skew hashing

Given parameters ℓ, L, partition the buckets into L sets.

Let S_i be the set of k-mers belonging to any bucket B_r with:

$$2^i < \left|B_r\right| < 2^{i+1} \text{ for } \ell < i < L$$

$$2^L < \left|B_r\right| \text{ for } i = L$$

For set S_i we need $\lceil \log_2(S_i) \rceil$ bits to write down an offset into a bucket of size $\left|S_i\right|$. Because of the skew distribution, we generally expect $\left|B_\ell\right| < \left|B_{\ell+1}\right| + \ldots + \left|B_L\right|$. So this skew hashing setup uses fewer bits for buckets that require fewer bits.

Querying with the "skew index"

Let begin $=$ ***sizes***[$f(r)$], end $=$ **sizes**[$f(r)$ + 1]

And let $i = \log(\text{end} - \text{begin}) - \ell$

If $i < 0$, then do the usual query.

Otherwise, let $q = P_i[f_i(g)]$,

and look at the super-kmer at ***offsets***[begin + q] on ***useq***

How to handle buckets with large $|B_r|$

1. Let A be the k-mers in buckets with size > 2^ℓ
2. Build an MPHF, h(.) over A
3. Store a vector P, with length |A|
4. At query time, for a queried k-mer g
5. P[h(g)] = q, says that g occurs on the q-th super-k-mer for the bucket that g belongs to.

Streaming Queries

Arguably the most critical optimization for “streamed” queries.

AAGCAACTGGT

AAGCAACTGGT

AAGCAACTGGT

Implement the caching scheme where, we simply save:

1. The position of the last hit
2. The offsets for B_r given that the last query had minimizer, r.

A note on double-strandedness

In the “*regular*” flavor of SSHash described so far… to handle double-strandedness, we query for both g and its reverse complement.

Or… in a *canonical* SSHash, a minimizer for g is defined as the min of the minimizers for g and $\bar{g}$.

How this is implemented and how this affects the implementation and properties of super k-mers is not really discussed in the paper.

Experiments – the data

Table 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

*paths are unipaths from SPSS decomposition. But this doesn’t matter too much for our purposes…

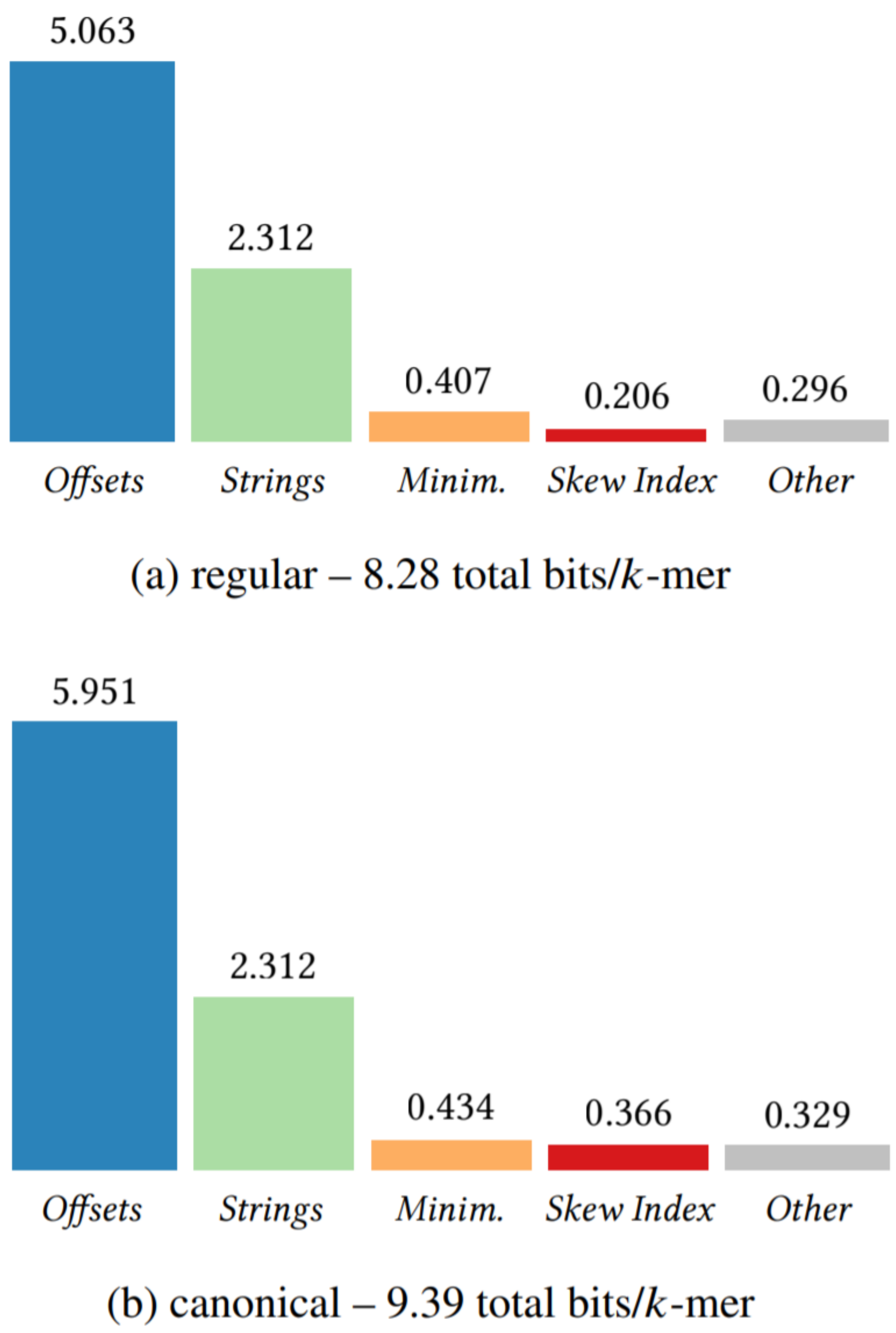


Fig. 1. Space breakdowns for the Human dataset, for both regular (a) and canonical (b) dictionaries. The numbers above each bar indicate the bits/k-mer spent by the respective components.

Table 4. 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	18.12	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

It's worth noting here that pufferfish stores information that supports queries that are more than ***just*** lookup(g). And can do more than just an MPHF…

Table 6. 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

Some observations about SSHash

1. Skew-hashing approach for building small exact data structures for the tail of a distribution is interesting.
2. The streaming workload significantly favors SSHash.
   - Other optimizations in this vein seem interesting.
3. SShash is a state-of-the-art associative container for k-mers, but is only the “sequence” part of the index. For a full reference index, you still need to pair it with an appropriate unitig -> reference mapping (more to come).