Improving indexing of the (compacted) colored de Bruijn graph

Problem & Motivation

K-mer based reference indexing

- Given a collection of reference sequences $\mathscr{R} = \{R_1, \ldots, R_m\}$, where each R_i is a string over the DNA alphabet $\Sigma = \{A, C, G, T\}$
- We want an index $\mathscr{I}$ over $\mathscr{R}$ that can efficiently answer the following queries:
  - **Membership:** *Does x appear in $\mathscr{R}$?*
  - **Count:** *How many times does x appear in $\mathscr{R}$?*
  - **Color:** *In which references does x appear?*
  - **Locate:** *Where in $\mathscr{R}$ does x appear?*

Increasing power / specificity of query.

- ***Applications :*** This type of index is useful for many foundational problems like read mapping/alignment/ lightweight alignment/pseudoalignment. Solving it quickly and in small space can help in bottleneck steps in taxonomic assignment, metagenomics, bulk and single-cell RNA-seq processing, etc.

Problem & Motivation

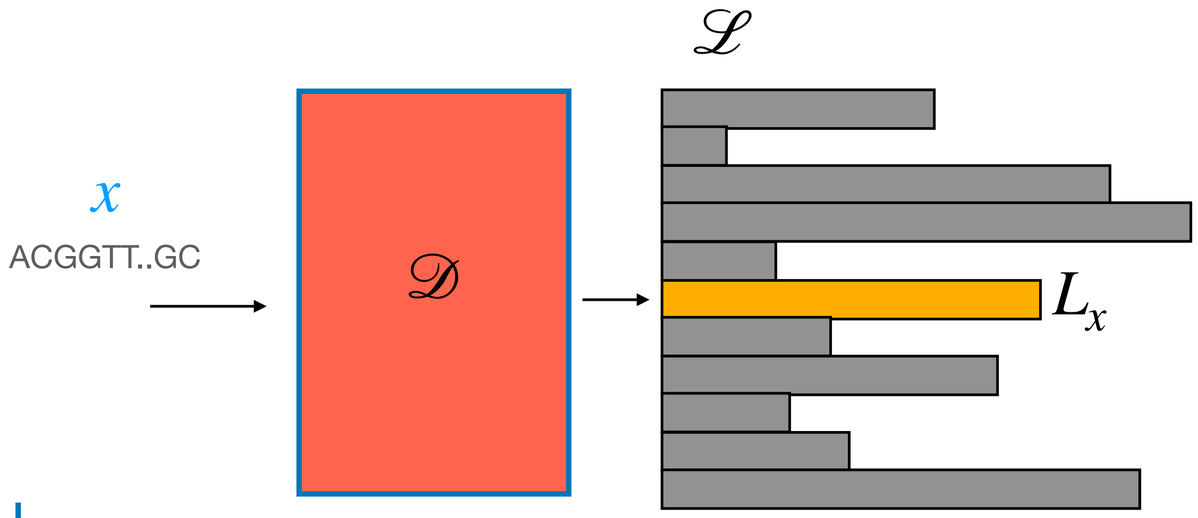
More formally

- Want: Map from distinct k-mers to their reference positions (i.e. for k-mer x):
  $x \rightarrow L_x = \{(i, \{p_{ij}\}), x \in R_i\}$

- Where $\mathscr{R} = \{R_1, \ldots, R_m\}$ is a set of m references, and L_x is a list of pairs of reference id i, and a ***set*** of occurrences of k-mer x on R_i

- Queries:

  - **Membership:** *Does x appear in $\mathscr{R}$?*
    Presence/absence in map
  - **Count:** *How many times does x appear in $\mathscr{R}$?*
    Length of L_x in the index
  - **Color:** *In which references does x appear?*
    The set $\{i \mid x \in R_i\}$
  - **Locate:** *Where in $\mathscr{R}$ does x appear?*
    The set $\{(i, \{p_{ij}\}) \mid x \in R_i\}$

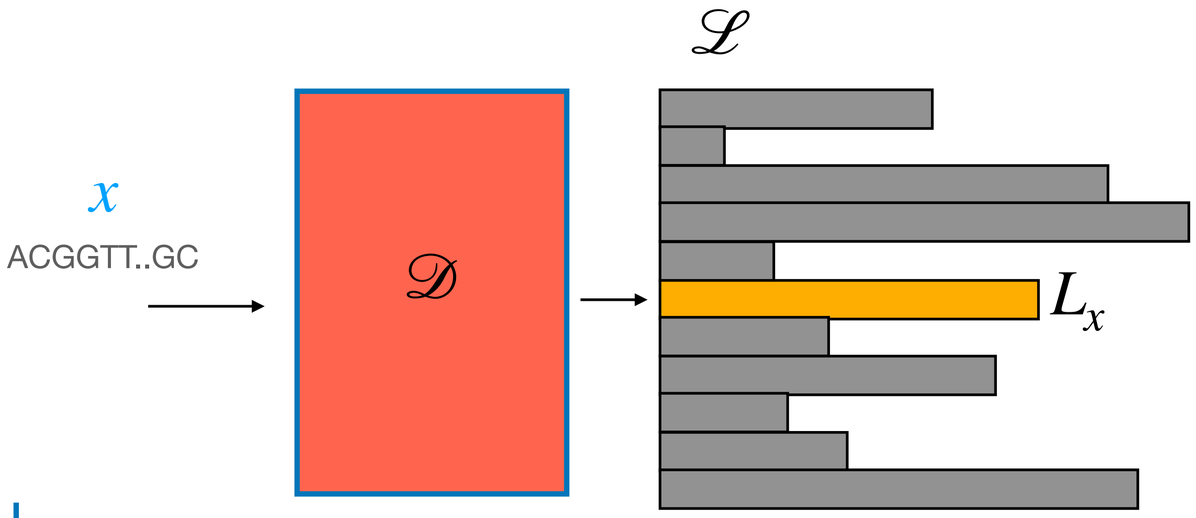


Increasing power / specificity of query.

Problem & Motivation

More formally

- Want: Map from distinct k-mers to their reference positions (i.e. for k-mer x):
  $x \rightarrow L_x = \{(i, \{p_{ij}\}), x \in R_i\}$
- Where $\mathscr{R} = \{R_1, \ldots, R_m\}$ is a set of m references, and L_x is a list of pairs of reference id i, and a ***set*** of occurrences of k-mer x on R_i
- Queries:
  - **Membership:** *Does x appear in $\mathscr{R}$?*
    Presence/absence in map
  - **Count:** *How many times does x appear in $\mathscr{R}$?*
    Length of L_x in the index
  - **Color:** *In which references does x appear?*
    The set $\{i \mid x \in R_i\}$
  - **Locate:** ***Where in $\mathscr{R}$ does x appear?***
    The set $\{(i, \{p_{ij}\}) \mid x \in R_i\}$

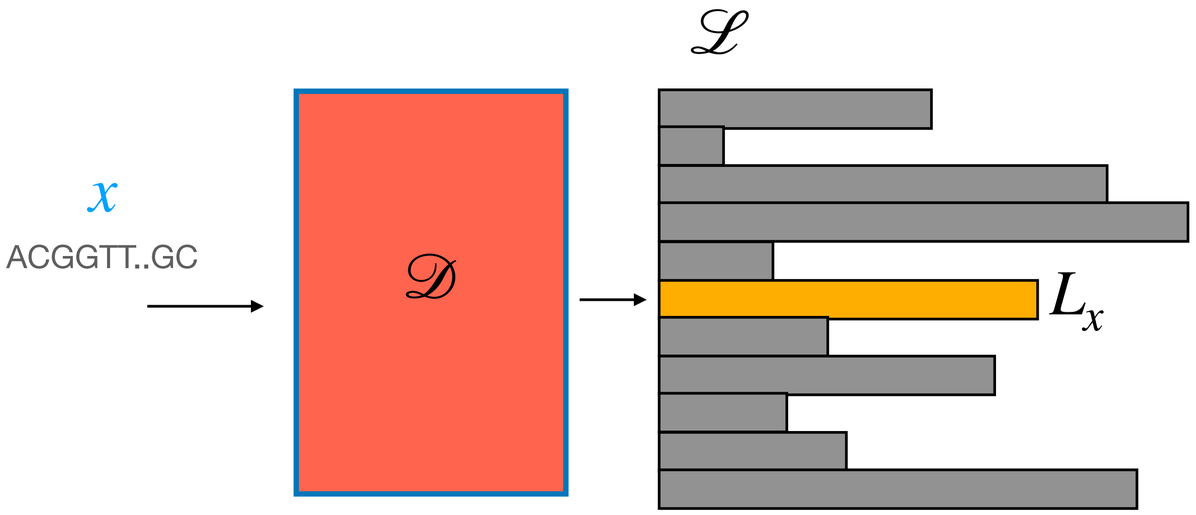


Increasing power / specificity of query.

Will focus on this as the most powerful / difficult query

A general structure for k-mer indexing

- Many k-mer based indexes are incarnations/adaptations of this **general indexing framework**, $\mathscr{D} + \mathrm{L}$:
  - deBGA [Liu et al. 2016]
  - Sequence Bloom Trees [Solomon et al. 2016]
  - kallisto [Bray et al. 2016]
  - BIGSI [Bradley et al. 2017]
  - Rainbowfish [Almodaresi et al. 2017]
  - Mantis [Pandey et al. 2018]
  - Pufferfish [Almodaresi et al. 2018]
  - SeqOthello [Yu et al. 2018]
  - COBS [Bingmann et al. 2019]
  - Reindeer [Marchet et al. 2020]
  - Raptor [Seiler et al. 2021]
  - Metagraph [Karasikov et al. 2022]
  - NIQKI [Agret et al. 2022]
  - Pufferfish2 [Fan et al. 2022]
  - etc.



Data structures based on *k*-mers for querying large collections of sequencing data sets

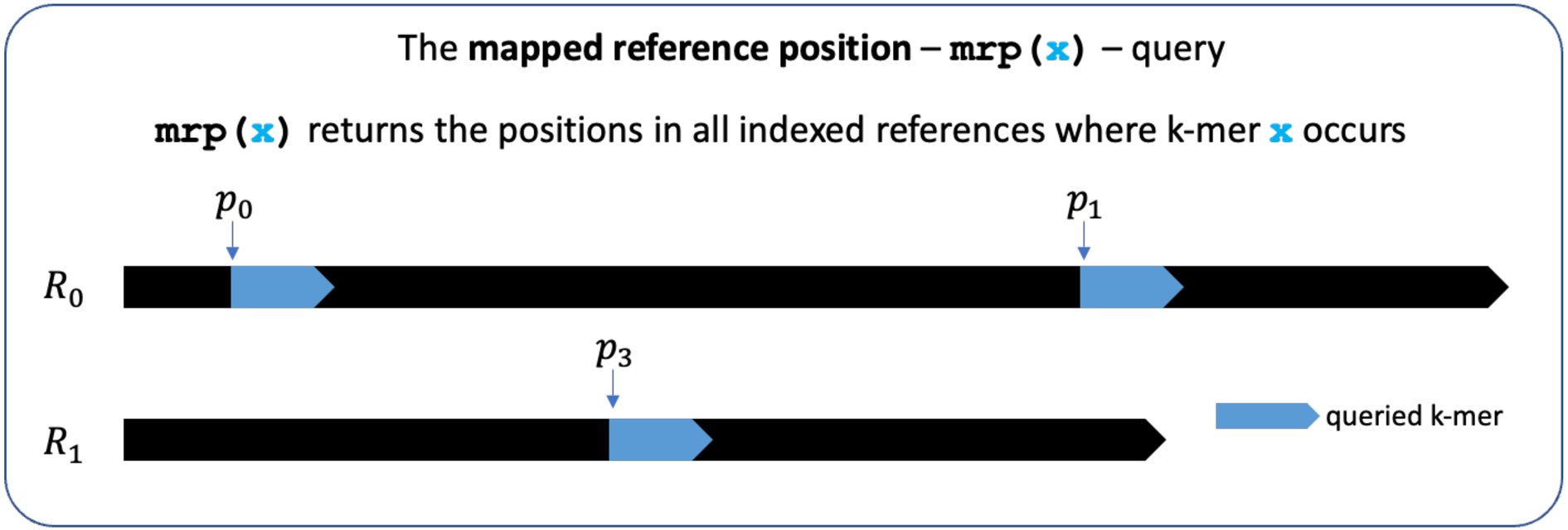
Camille Marchet[1], Christina Boucher[2], Simon J. Puglisi[3], Paul Medvedev[4,5,6], Mikaël Salson[1] and Rayan Chikhi[7]

Problem & Motivation

The fundamental query — mrp()

We want to find the position of any k-mer (e.g. **x**) in an index over **thousands or hundreds of thousands** of known reference sequences.

For example, when comparing observed sequences from the microbiome to known bacterial strains and species.



The (compacted colored) de Bruijn graph for reference indexing

Using the (compacted colored) de Bruijn graph for indexing

Goal: compactly represent input reference sequences

R_0: **AAATGAG**

R_1: **AAATGACG**

R_2: **CCTGACG**

R_3: **CCTGAG**

Constructing a *de Bruijn Graph*

1. Break references into k-mer set (e.g. k=3)
2. Join k-mers with (k-1) overlap

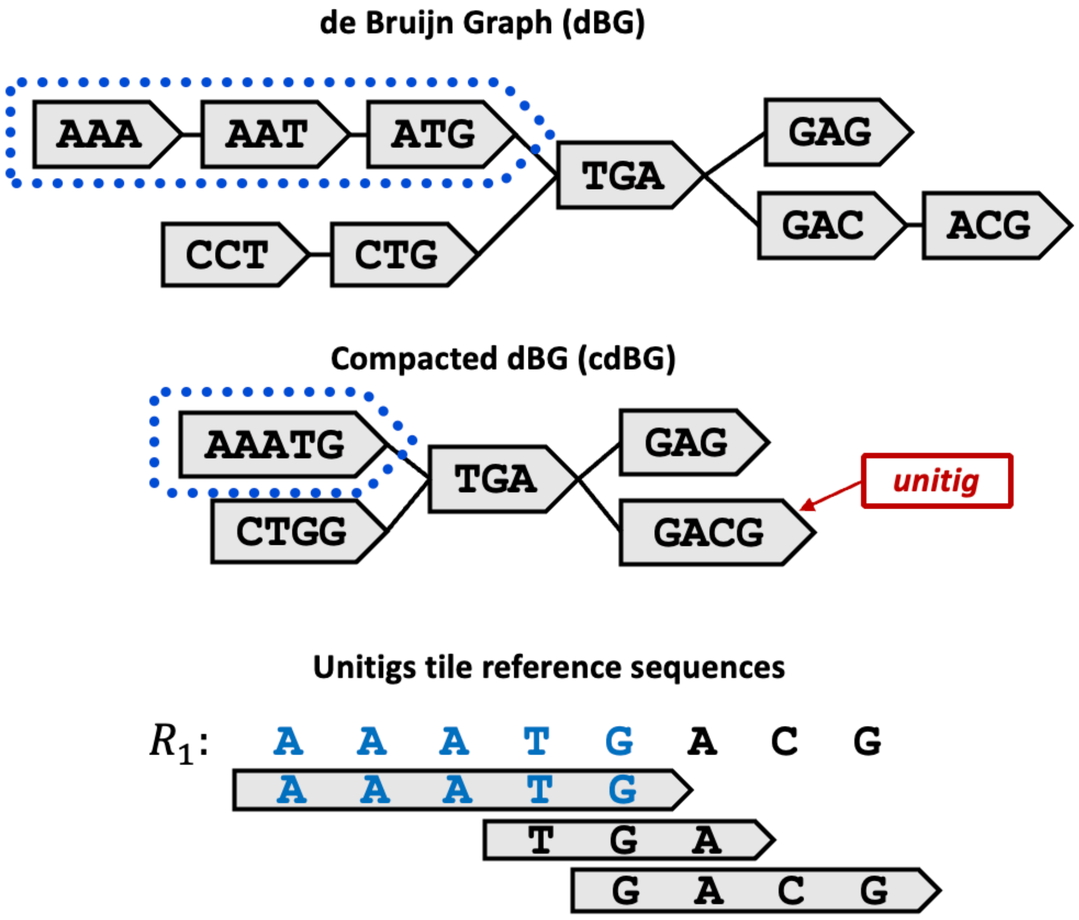
***Compacted* de Bruijn Graph**

Merge non-branching paths in dBG into *unitigs*

Key properties:

Unitigs *tile* reference sequences

Any k-mer occurs *exactly once*, in *one unique* unitig U_i



The reference index as a *composition* of 2 maps

k2tile(x) returns:

1. The ***identity*** of the unitig U_i that contains **x**
2. The ***offset*** *(position) into* U_i where **x** occurs

Achieved (in Pufferfish) by:

1. Storing the unitig sequences
2. Building a minimum perfect hash function over k-mers in an input reference collection

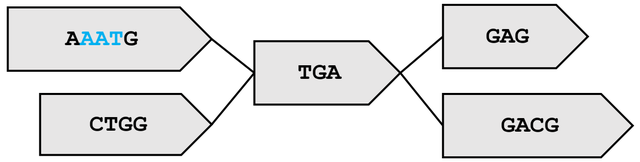
tile2occ(U_i) returns:

1. A list of tuples of ***(reference, position, orientation)*** triplets of the unitig U_i that contains **x**

Achieved (in Pufferfish) by:

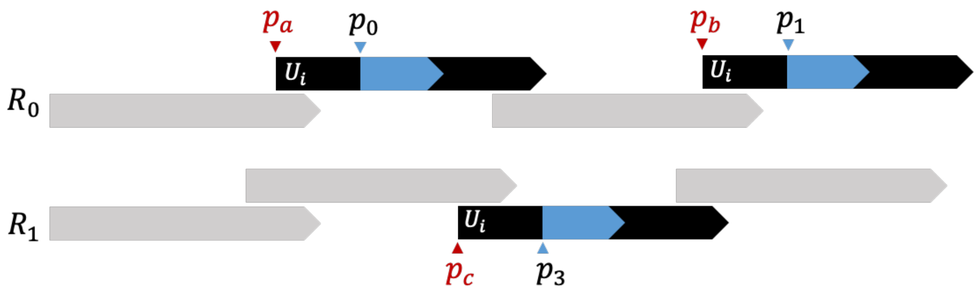
1. Storing a "flattened" inverted map of unitig ids to lists of occurrences (i.e. **utab()** on the right).

Compacted dBG (cdBG)



`k2tile(x: Kmer) -> (i, offset)`

`tile2occ(Ui: Unitig id) -> [(Ri, offsetij, oriij)…]`



utab

	…
U_i:	$(R_0, p_a), (R_0, p_b), (R_1, p_c), \ldots$
	…
	…
	…

We will not discuss the details in this presentation, but will need to know the inputs and outputs of `k2tile(…)`*, and that it is O(1).*

Why does indexing this way help?

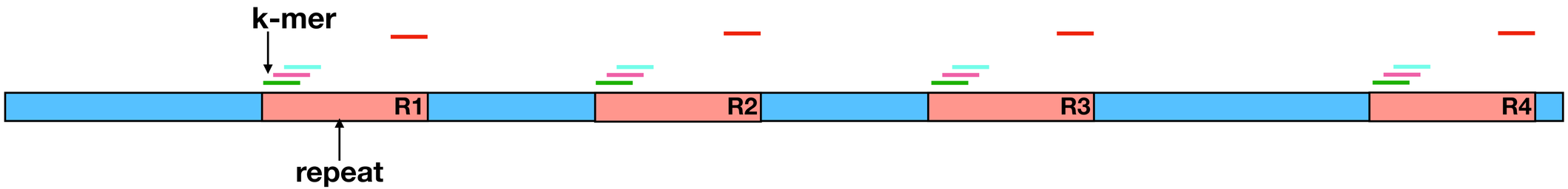
Compression through “factorization”

$$(2 \times 4) + (2 \times 5) + (3 \times 8) + (2 \times 6) + (3 \times 4) + (3 \times 5) = 2\,(4+5+6) + 3\,(8+4+5)$$

Why does indexing this way help?

Compression through “factorization”

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



List all occurrences individually

→ R_1-I_1, R_2 - I_1, …, R_M - I_1

→ R_1-I_1+1, R_2 - I_1+1, …, R_M - I_1+1

→ R_1-I_1+2, R_2 - I_1+2, …, R_M - I_1+2

⋮ …

→ R_1-k, R_2 - k, …, R_M - k

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

→ R_1-I_1, R_2 - I_1, …, R_M - I_1

→ 0

→ 1

→ 2

⋮

→ I_1-k

The cdBG removes redundancy by providing an extra level of indirection

What's the benefit of this "framework"?

- Recognizing the minimal API for such an index as the composition of these two maps (**k2tile(x)**, and **tile2occ(U_i)**) leads to a *modular indexing framework.*
- We can mix-and-match different data structures for each of the maps (e.g. use a MPHF, FM-index, r-index or something else for **k2tile(x)**).
- Allowed us to *immediately* capitalize on recent advancements in k-mer indexing and maps (replacing Pufferfish's **k2tile()** with sshash [Pibiri 2022]).

What's the benefit of this "framework"?

- Recognizing the minimal API for such an index as the composition of these two maps (**k2tile(x)**, and **tile2occ(U_i)**) leads to a *modular indexing framework.*
- We can mix-and-match different data structures for each of the maps (e.g. use a MPHF, FM-index, r-index or something else for **k2tile(x)**).
- Allowed us to *immediately* capitalize on recent advancements in k-mer indexing and maps (replacing Pufferfish's **k2tile()** with sshash [Pibiri 2022]).

Basic idea implemented in *piscem* 🐟:

COMBINE-lab / **piscem** Public

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

Immediate benefits of piscem

- Prior to piscem, **pufferfish** has been a state-of-the-art (in terms of size & speed) hash-based (very fast) ccDBG index.

	Pufferfish	**Pufferfish (sparse)**	**piscem**
Human *"splici" index*	7.7G	5.2G	2.5G
GRCh38	15.2G	10.1G	4.7G
7 human	36G	28G	12G

- Speed is *fast* but somewhat (30-40%) slower than pufferfish.
- Can map 638M reads (10x PBMC 10k dataset) in 18 minutes using 16 threads.

Improving the tile2occ() map (the bottleneck)

One of these things is not like the other

1 human genome

7 human genomes

- sshash makes a great **k2tile()** map, but as we index more sequence, the **tile2occ()** map becomes the clear bottleneck.
- **k2tile()** grows in the amount of “unique” sequence, while **tile2occ()** grows (at least) in the total reference length.
- How can we compress **tile2occ()** and keep access fast?

A new scheme for representing tilings

Spectrum Preserving Tilings Enable Sparse and Modular Reference Indexing

Jason Fan[1], Jamshed Khan[1], Giulio Ermanno Pibiri[2,3], and Rob Patro[1](✉)

[1] University of Maryland, College Park, MD 20440, USA
jasonfan@umd.edu, {jamshed,rob}@cs.umd.edu
[2] Ca' Foscari University of Venice, Venice, Italy
giulioermanno.pibiri@unive.it
[3] ISTI-CNR, Pisa, Italy

https://doi.org/10.1007/978-3-031-29119-7_2

3.1 Definition

Given a k-mer length k and an input reference collection of genomic sequences $\mathcal{R} = \{R_1, \dots, R_N\}$, a spectrum preserving tiling (SPT) for $\mathcal{R}$ is $\Gamma := (\mathcal{U}, \mathcal{T}, \mathcal{S}, \mathcal{W}, \mathcal{L})$:

- **Tiles**: $\mathcal{U} = \{U_1, \dots, U_F\}$. The set of *tiles* is a spectrum preserving string set, i.e., a set of strings such that each k-mer in $\mathcal{R}$ occurs in some $U_i \in \mathcal{R}$. Each string $U_i \in \mathcal{U}$ is called a *tile*.
- **Tiling sequences**: $\mathcal{T} = \{T_1, \dots, T_N\}$ where each T_n corresponds to each reference $R_n \in \mathcal{R}$. Each tiling sequence is an ordered sequence of tiles $T_n = [T_{n,1}, \dots, T_{n,M_n}]$, of length M_n, with each $T_{n,m} = U_i \in \mathcal{U}$. We term each $T_{n,m}$ a *tile-occurrence*.
- **Tile-occurrence lengths**: $\mathcal{L} = \{L_1, \dots, L_N\}$, where each $L_n = [l_{n,1}, \dots, l_{n,M_n}]$ is a sequence of lengths.
- **Tile-occurrence offsets**: $\mathcal{W} = \{W_1, \dots, W_N\}$, where each $W_n = [w_{n,1}, \dots, w_{n,M_n}]$ is an integer-sequence.
- **Tile-occurrence start positions**: $\mathcal{S} = \{S_1, \dots, S_N\}$, where each $S_n = [s_{n,1}, \dots, s_{n,M_n}]$ is an integer-sequence.

A valid SPT must satisfy the *spectrum preserving tiling property*, that every reference sequence R_n can be reconstructed by gluing together *substrings of tiles* at offsets W_n with lengths L_n:

$$R_n = T_{n,1}[w_{n,1} : w_{n,1} + l_{n,1}] \oplus \dots \oplus T_{n,M_n}[w_{n,M_n} : w_{n,M_n} + l_{n,M_n}].$$

A new scheme for representing tilings

Spectrum Preserving Tilings Enable Sparse and Modular Reference Indexing

Jason Fan[1], Jamshed Khan[1], Giulio Ermanno Pibiri[2,3], and Rob Patro[1](✉)

[1] University of Maryland, College Park, MD 20440, USA
jasonfan@umd.edu, {jamshed,rob}@cs.umd.edu
[2] Ca' Foscari University of Venice, Venice, Italy
giulioermanno.pibiri@unive.it
[3] ISTI-CNR, Pisa, Italy

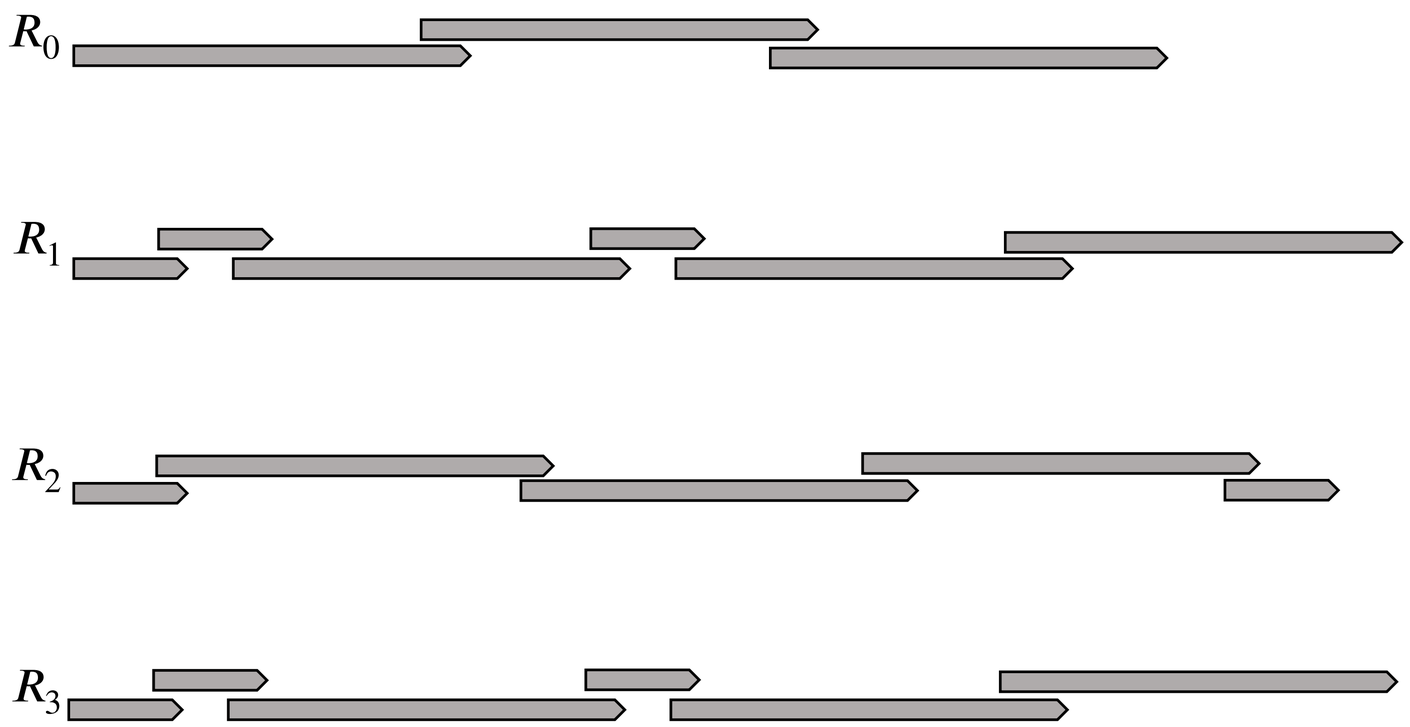
https://doi.org/10.1007/978-3-031-29119-7_2

The full details are not important for the purpose of this lecture, but there is a fully fleshed out theory for these composable indices based on the novel idea of Spectrum Preserving Tilings (SPTs).

It is *very general*. You need not use unitigs, but could use e.g. simplitigs, eulertigs, etc.

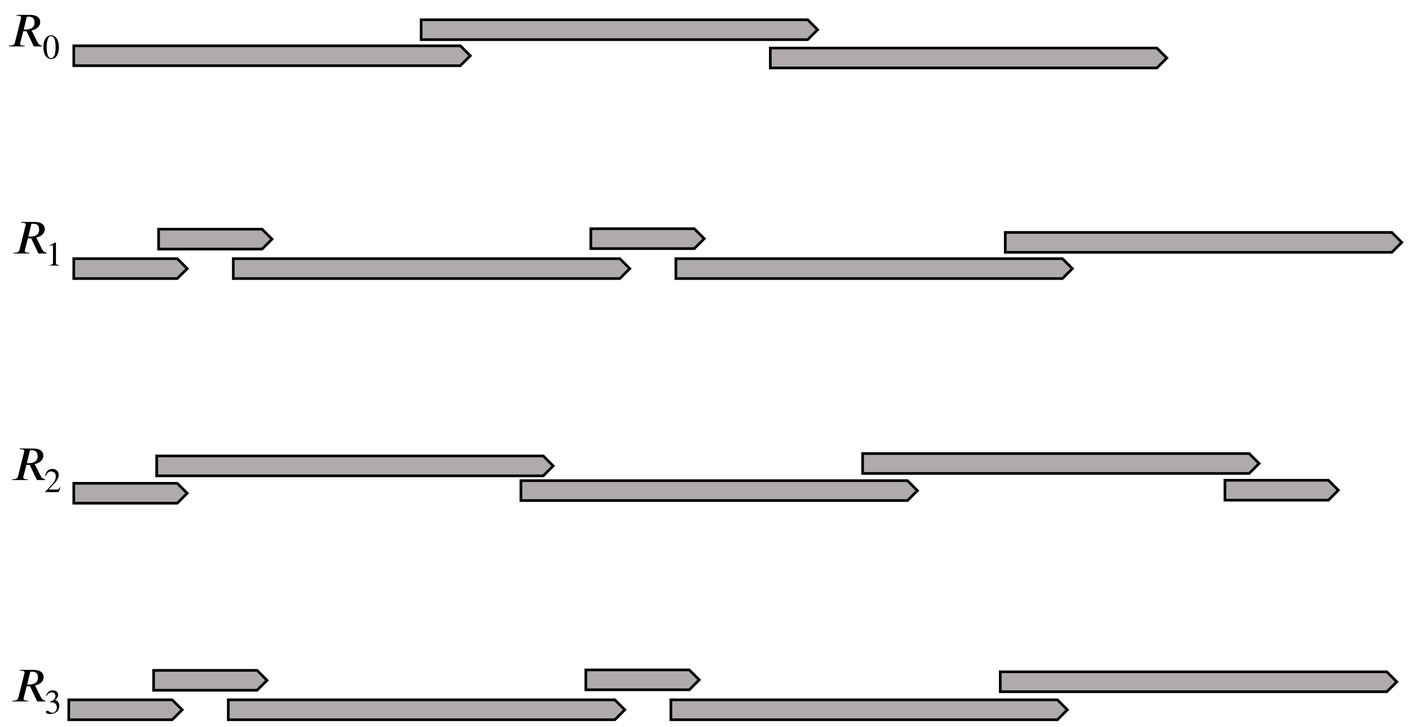
Focusing on tile2ref()

How do we compress the bottleneck component?



Focusing on tile2ref()

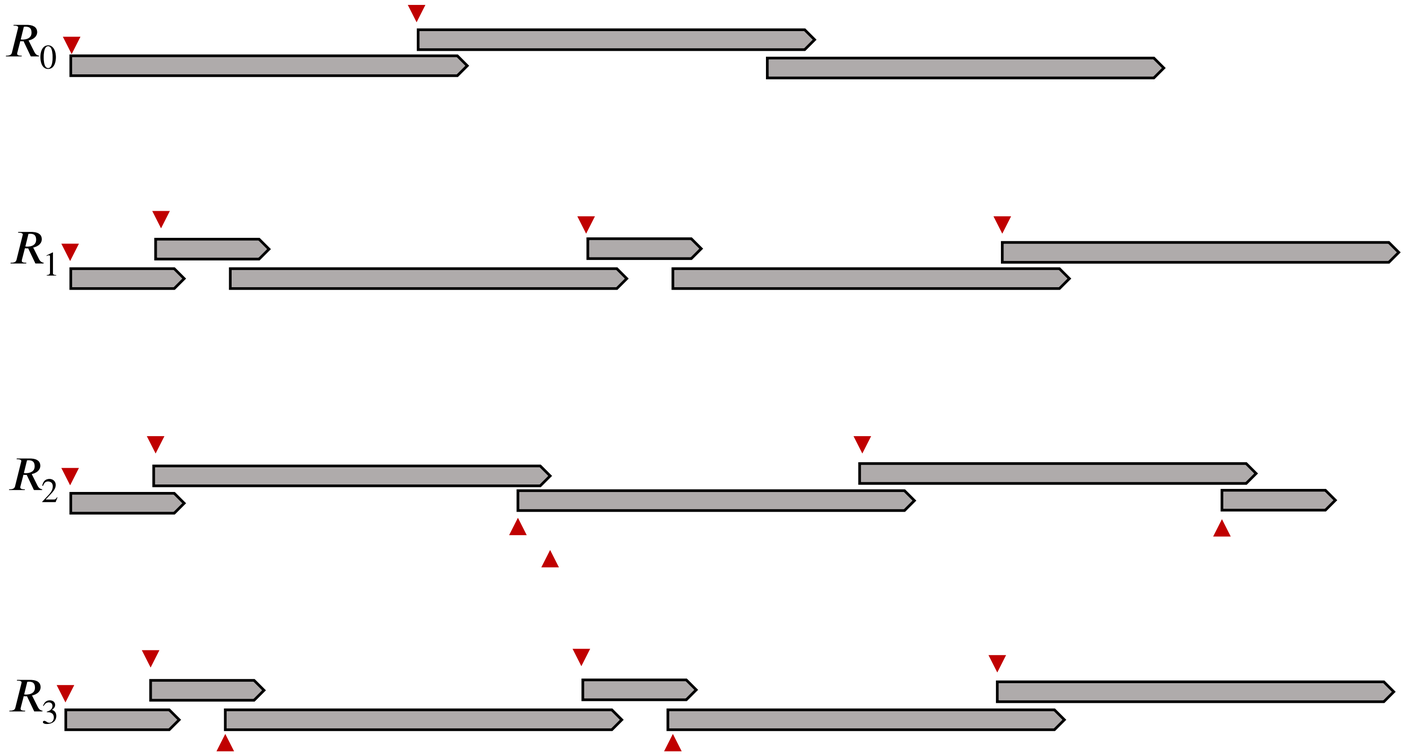
How do we compress the bottleneck component?



`ctab` needs **log(...)** bits per occurrence for each unitig.

Focusing on tile2ref()

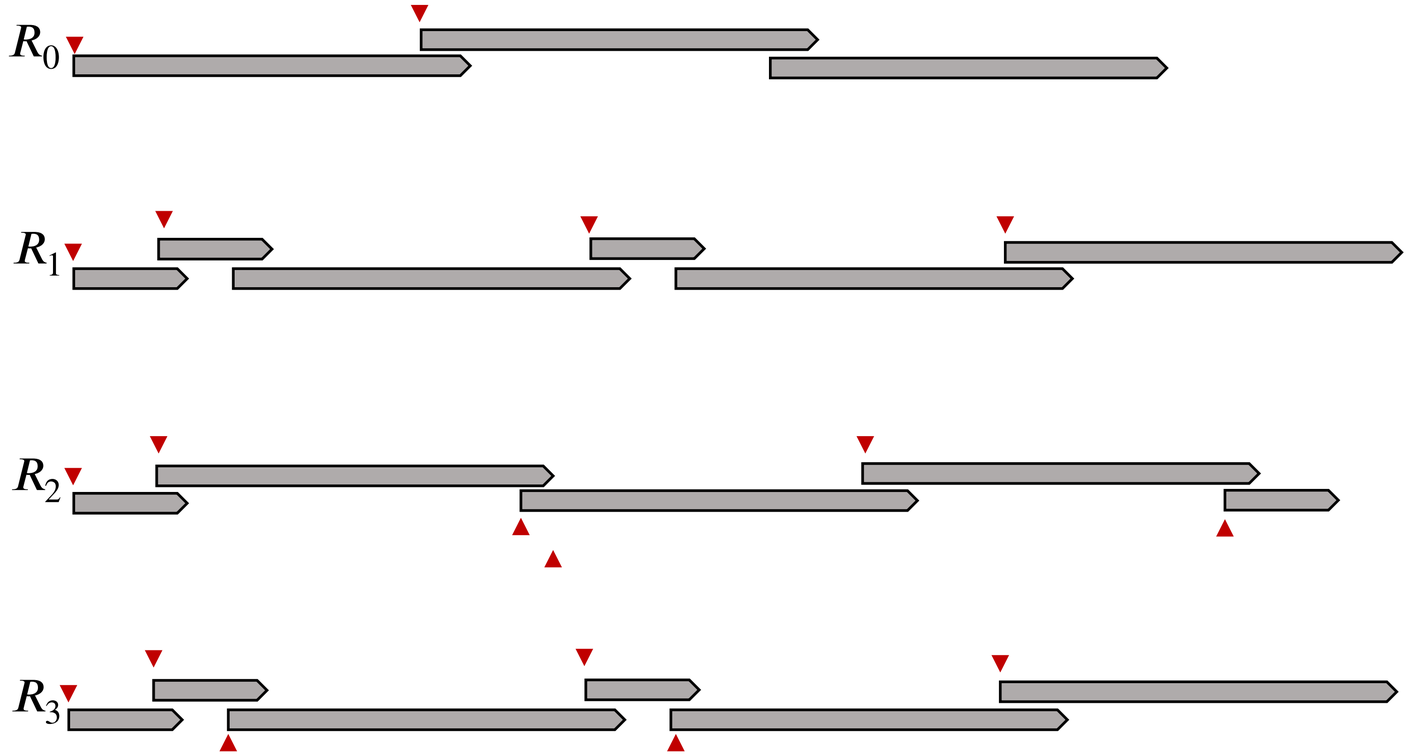
How do we compress the bottleneck component?



`ctab` needs **log(...)** bits per occurrence for each unitig.

Focusing on tile2ref()

How do we compress the bottleneck component?

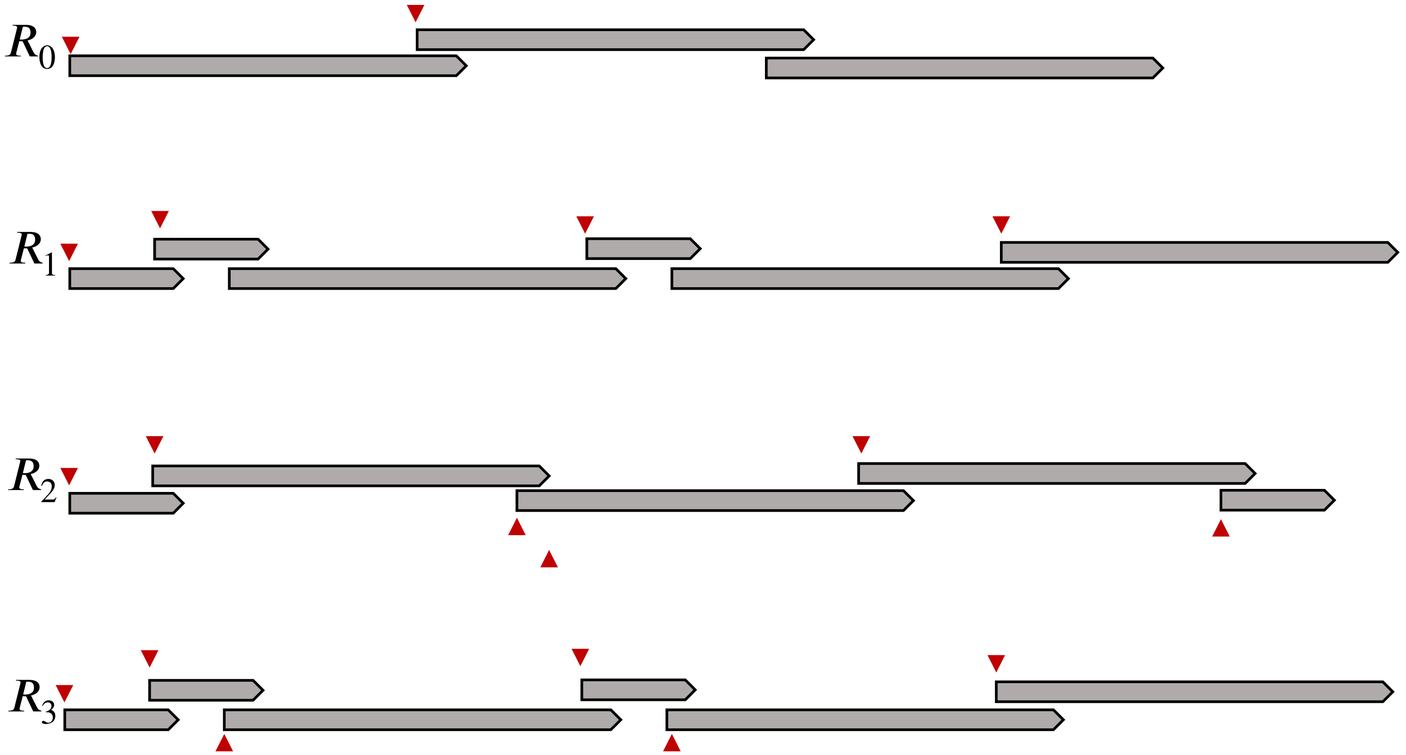


`ctab` needs **log(…)** bits per occurrence for each unitig.

Can we store log(…) bits per occurrence only for some unitigs?

Focusing on tile2ref()

How do we compress the bottleneck component?



`ctab` needs **log(…)** bits per occurrence for each unitig.

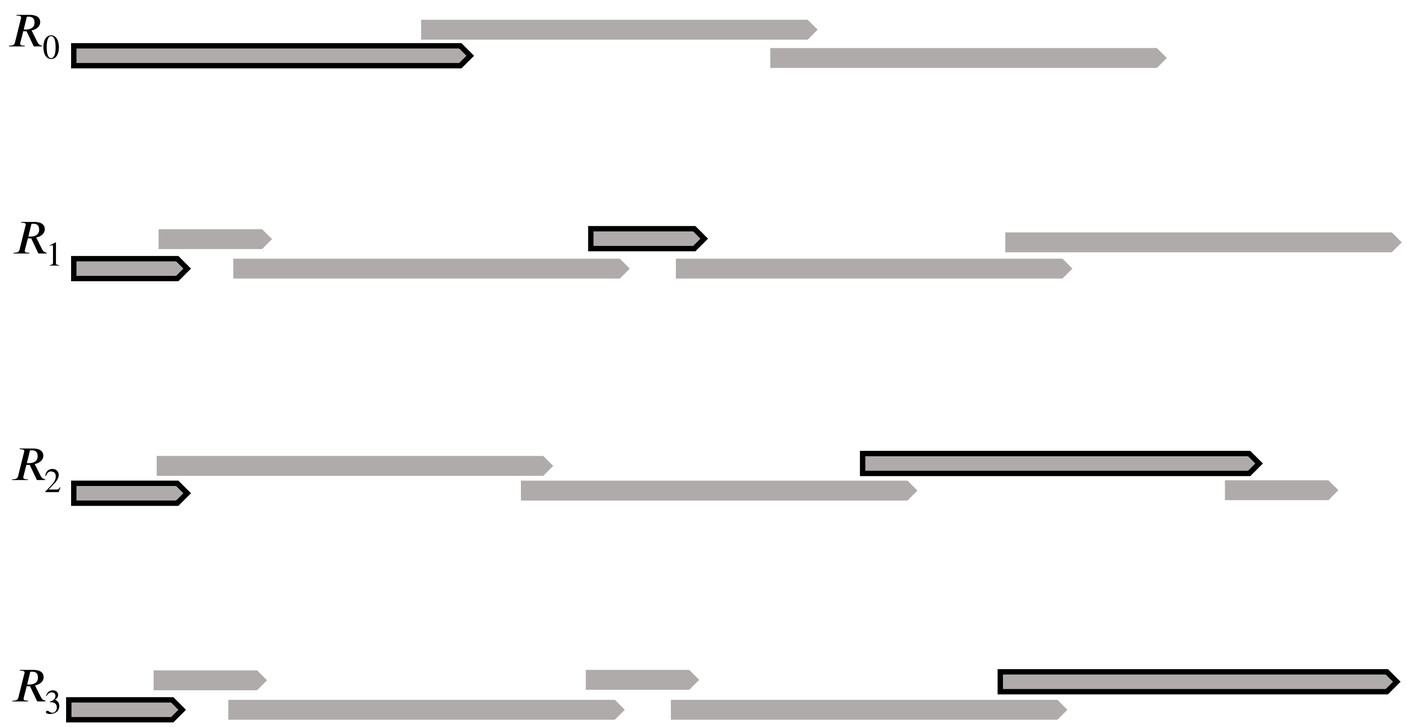
Can we store log(…) bits per occurrence only for some unitigs?

Yes!

Focusing on tile2ref()

How do we compress the bottleneck component?

samples a subset of unique unitigs to compress **`utab`**:



`ctab` needs **log(…)** bits per occurrence for each unitig.

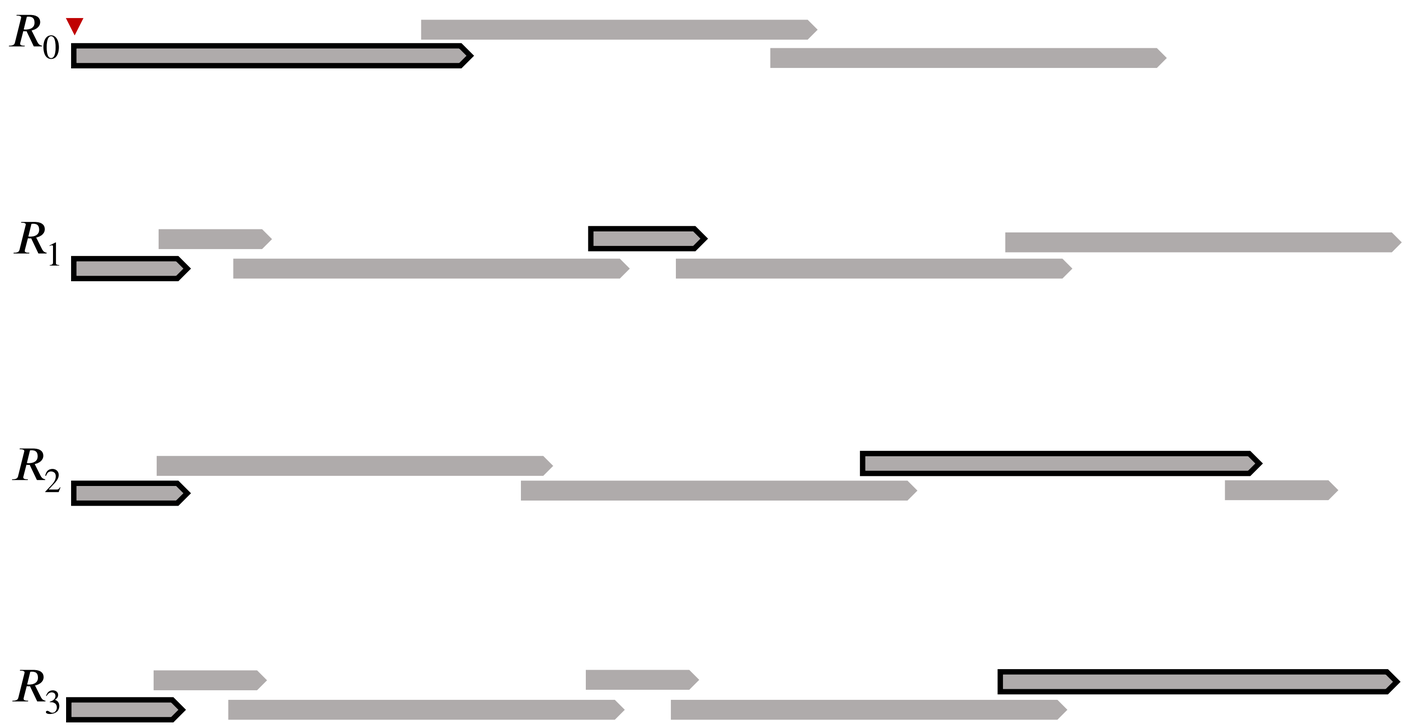
Can we store log(…) bits per occurrence only for some unitigs?

Yes!

Focusing on tile2ref()

How do we compress the bottleneck component?

samples a subset of unique unitigs to compress **`utab`**:



`ctab` needs **log(…)** bits per occurrence for each unitig.

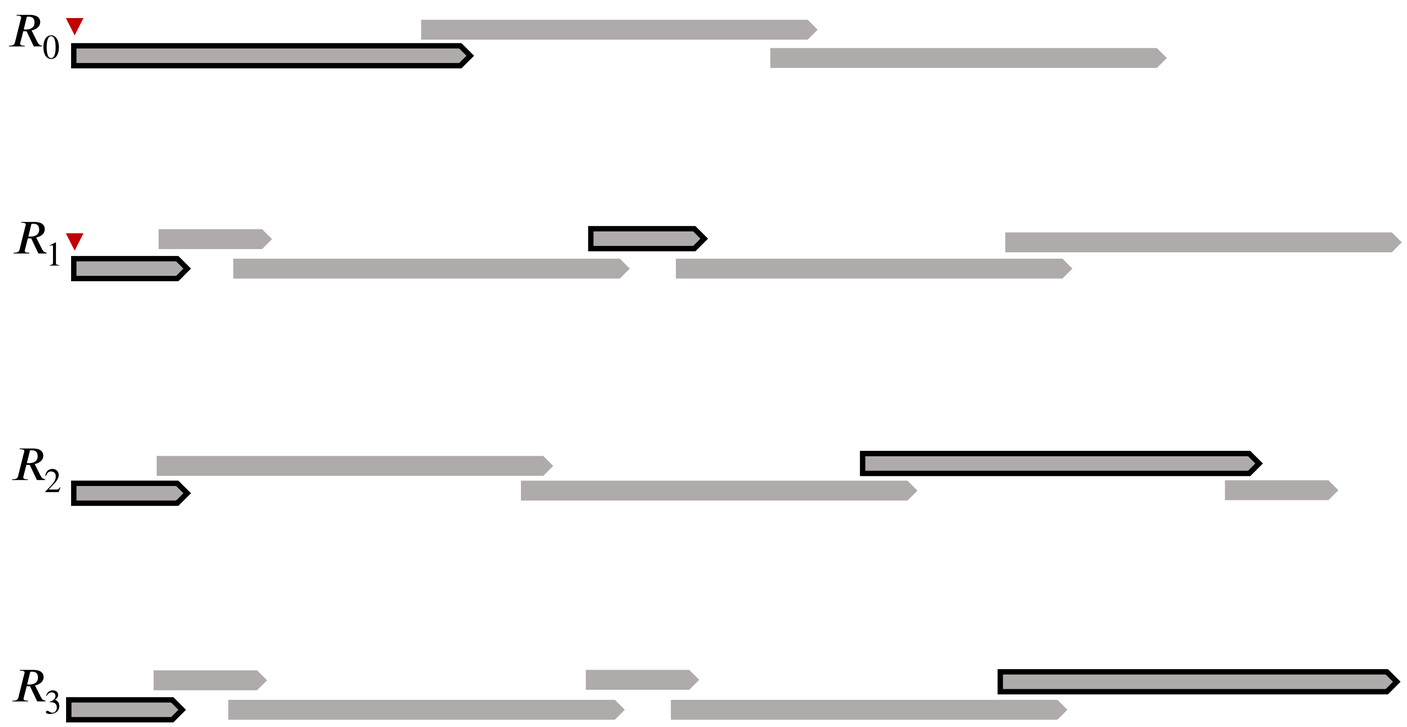
Can we store log(…) bits per occurrence only for some unitigs?

Yes!

Focusing on tile2ref()

How do we compress the bottleneck component?

samples a subset of unique unitigs to compress **`utab`**:



`ctab` needs **log(…)** bits per occurrence for each unitig.

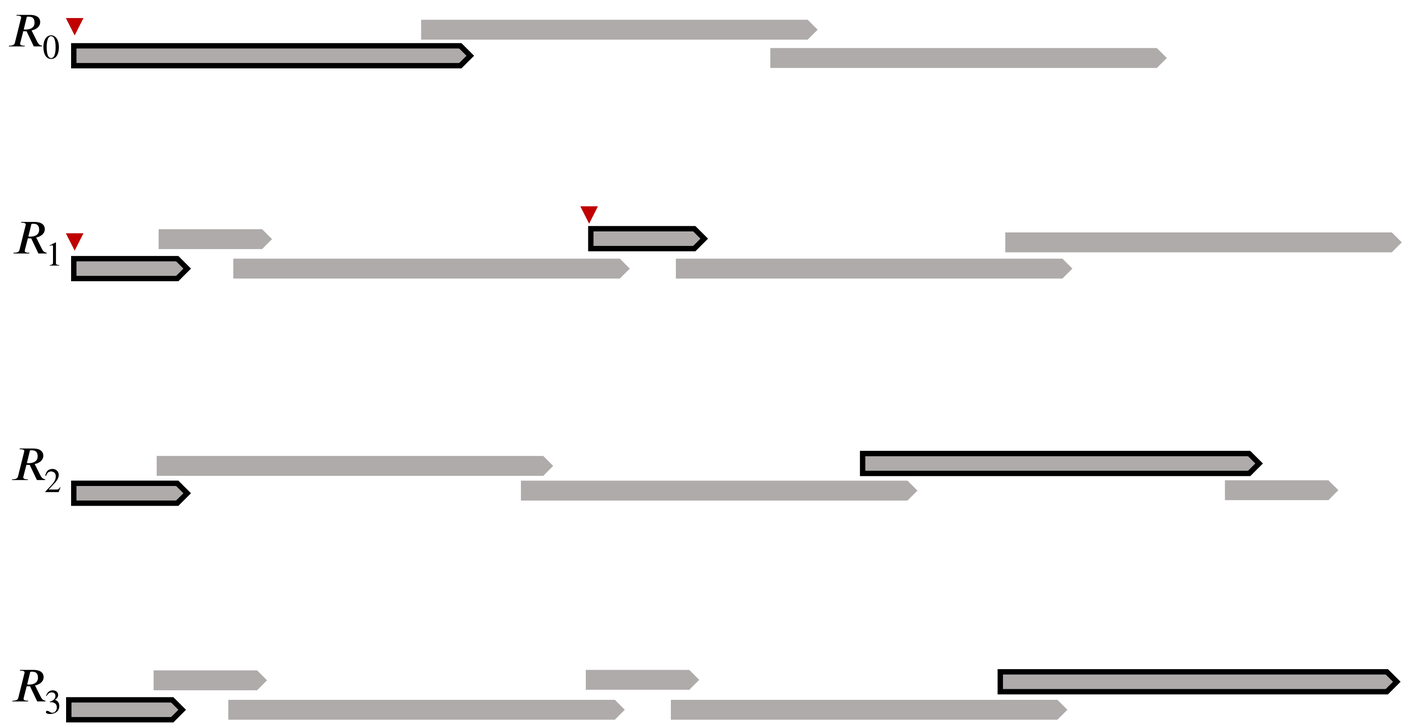
Can we store log(…) bits per occurrence only for some unitigs?

Yes!

Focusing on tile2ref()

How do we compress the bottleneck component?

samples a subset of unique unitigs to compress **`utab`**:



`ctab` needs **log(…)** bits per occurrence for each unitig.

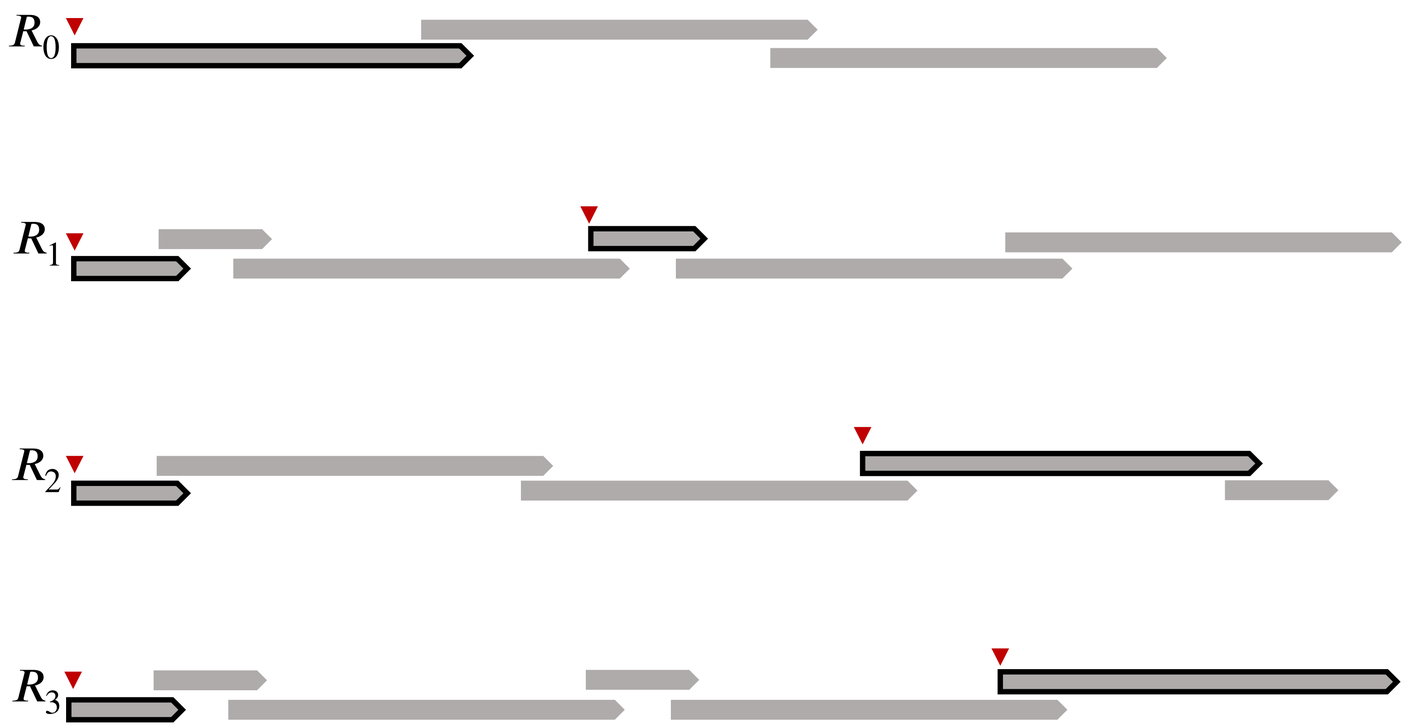
Can we store log(…) bits per occurrence only for some unitigs?

Yes!

Focusing on tile2ref()

How do we compress the bottleneck component?

samples a subset of unique unitigs to compress **`utab`**:



`ctab` needs **log(…)** bits per occurrence for each unitig.

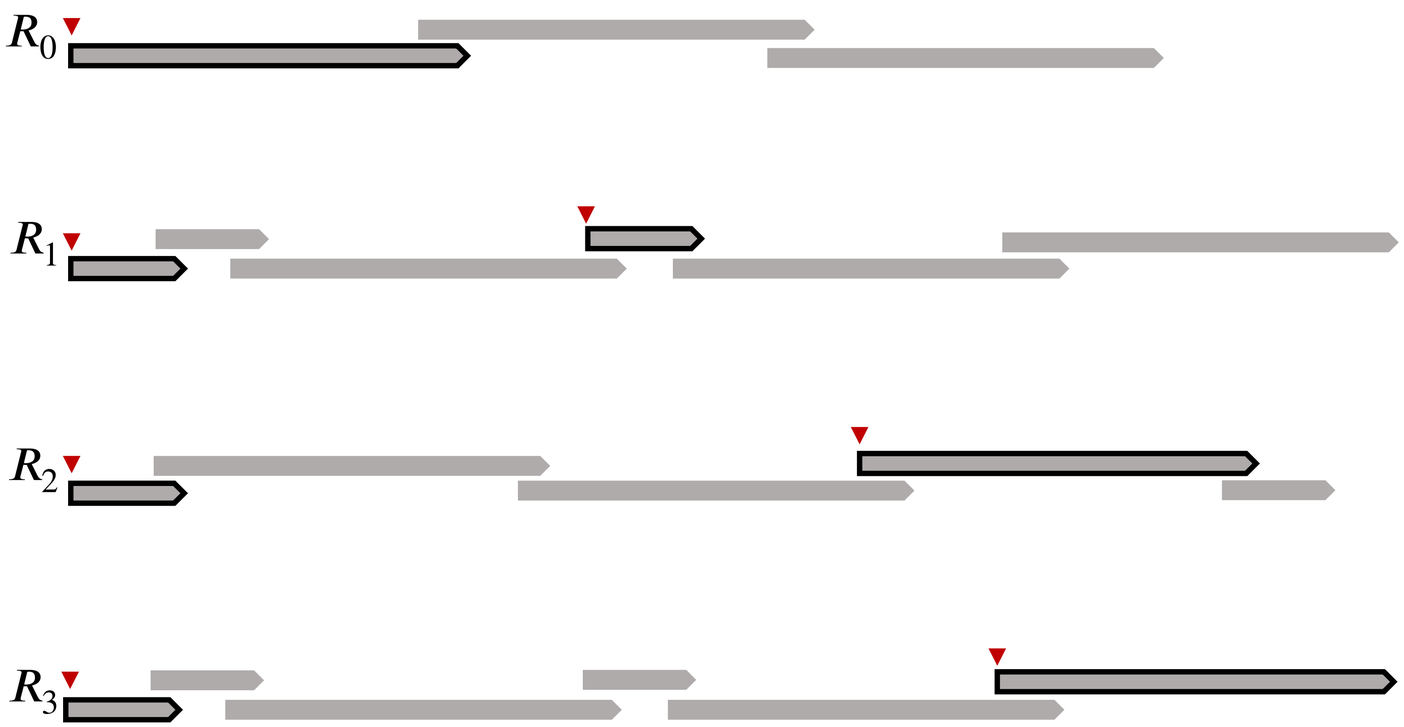
Can we store log(…) bits per occurrence only for some unitigs?

Yes!

Focusing on tile2ref()

How do we compress the bottleneck component?

samples a subset of unique unitigs to compress **`utab`**:



`ctab` needs **log(…)** bits per occurrence for each unitig.

Can we store log(…) bits per occurrence only for some unitigs?

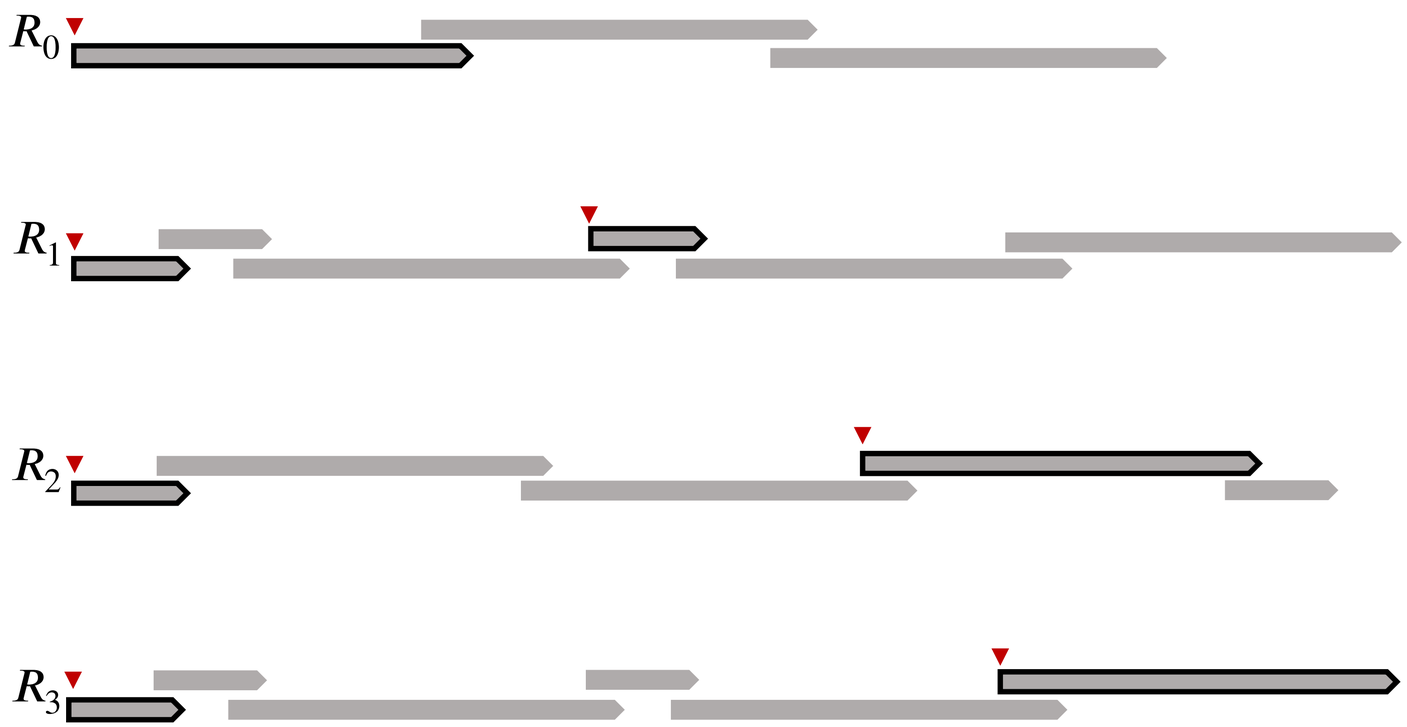
Yes!

Sampled – **log(L)** bits per occ.

Focusing on tile2ref()

How do we compress the bottleneck component?

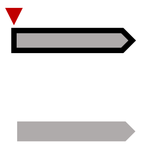
samples a subset of unique unitigs to compress **`utab`**:



`ctab` needs **log(…)** bits per occurrence for each unitig.

Can we store log(…) bits per occurrence only for some unitigs?

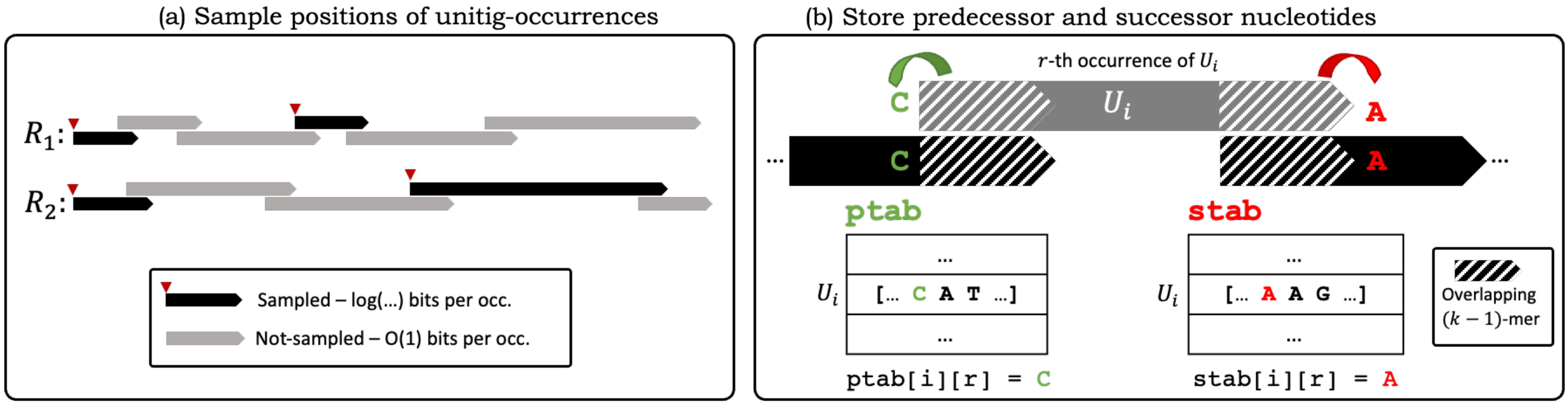
Yes!



Sampled – **log(L)** bits per occ.

Not-sampled – ~**7** bits per occ.

For non-sampled unitigs, store predecessor nucleotides

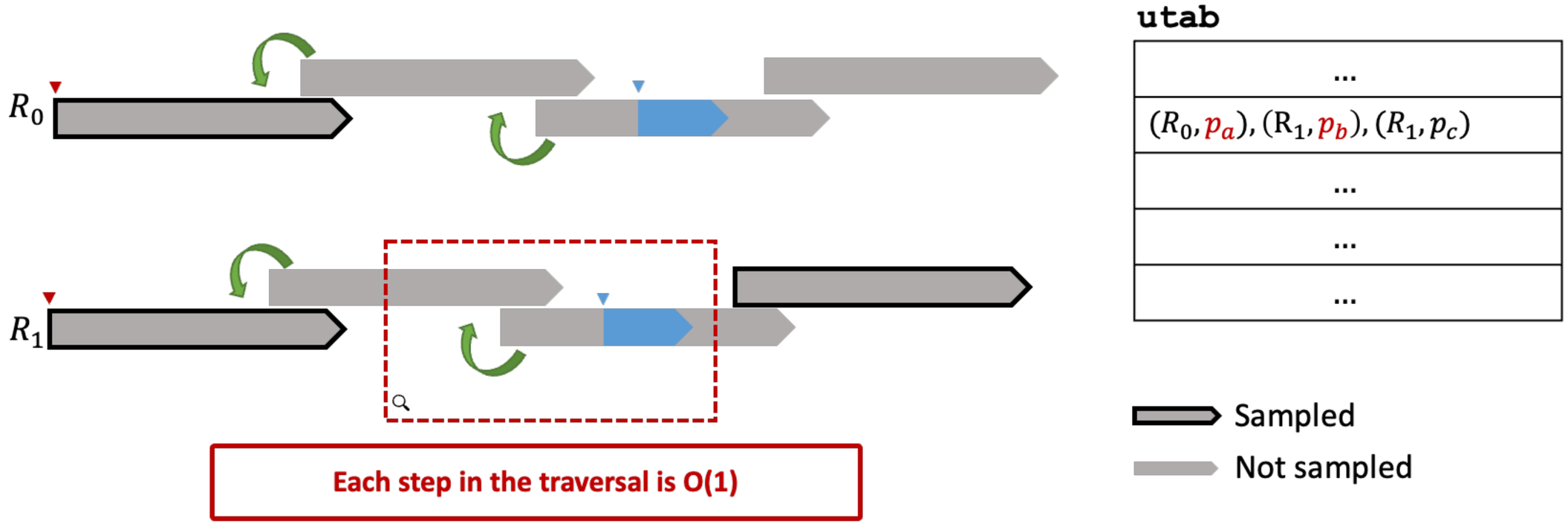


Need to store the predecessors **&** successors, because each backward step needs to determine *which* occurrence of the predecessor unitig is being traversed.

Efficient access to specific predecessor / successor nucleotides via rank and select over Σ in O(1) time using the wavelet matrix [Claude et al. 2013]

Querying non-sampled unitigs

- Traverse backward toward the closest sampled unitig
- Inferring the position of the non-sampled unitig is trivial to infer (sampled position + distance walked)



- Note: A naive implementation of this idea is asymptotically optimal, but practically slow — much engineering goes into making this *practically* fast.

This sampling scheme lets us shrink tile2occ()

We can *explicitly* trade off size for speed

Dataset	Sampling strategy	Index size (GB)	100K reads (secs)
7 Humans	None	16.8	139.4
	Random ($s = 3, t = .05$)	7.8 (2.15×)	8092.8 (58.04×)
	Random ($s = 3, t = .25$)	9.9 (1.70×)	1466.2 (10.52×)
4000 *E. coli*	None	7.7	12.6
	Random ($s = 3, t = .05$)	3.7 (2.08×)	15.6 (1.24×)
	Random ($s = 3, t = .25$)	4.7 (1.63×)	15.5 (1.23×)
30K Human gut	None	86.3	178.7
	Random ($s = 3, t = .05$)	45.6 (1.90×)	570.2 (3.19×)
	Random ($s = 3, t = .25$)	54.4 (1.59×)	576.9 (3.23×)
	Random ($s = 6, t = .05$)	34.6 (2.50×)	644.8 (3.61×)
	Random ($s = 6, t = .25$)	45.6 (1.90×)	646.1 (3.56×)

Smaller indices make indexing larger sequence possible

And save 💵

Dataset	u2occ with pufferfish2	k2u with SSHash	New index	Original pufferfish index
7 Human	9.9	3.2	**13.1**	28.0
4000 *E. coli*	3.7	7.3	**11.0**	26.1
30K Human gut	34.6	22.0	**55.6**	131.7

AWS EC2 instances pricing:

https://instances.vantage.sh/aws/ec2/x2gd.xlarge

64 GiB of RAM — **243 USD** per month

https://instances.vantage.sh/aws/ec2/x2gd.2xlarge

128 GiB of RAM — **478 USD** per month

https://instances.vantage.sh/aws/ec2/x2gd.4xlarge

256 GiB of RAM — **975 USD** per month

Conclusions

- The *reference indexing problem* admits a modular solutions made up of two distinct abstract data types: a **dictionary** $\mathcal{D}$ (k2tile) and an **inverted index** $\mathcal{L}$ (tile2occ).
- While substantial work has been done on how to represent $\mathcal{D}$, relatively little work has been done on how to represent $\mathcal{L}$ (especially for genomic references).
- The **spectrum preserving tiling** formalism, and reasoning about reference tilings opens up the possibility of sampling tiling occurrences.
- Viewing the reference index as the modular composition of 2 distinct data structures, and making the necessary API explicit, opens the door to constructing a whole *class* of reference indexing data structures.

Some open problems

More in the paper

1. Can we use, or at least mix-and-match sampling with traditional compression techniques for inverted lists (Elias-Fans, interpolative encoding, etc.)?

2. We currently sample entire *unitigs* (i.e. all occurrences) — what if we sample specific occurrences instead?

3. What is the best set of tiles? We used unitigs, but simplitigs, eulertigs, etc. are possible. It is *not* obvious that longer “tigs” → smaller representations.

4. We considered only *exact* / *lossless* indexing, but what could we achieve if we allow approximation? E.g. do not index all tiles or allow some false-positive results.