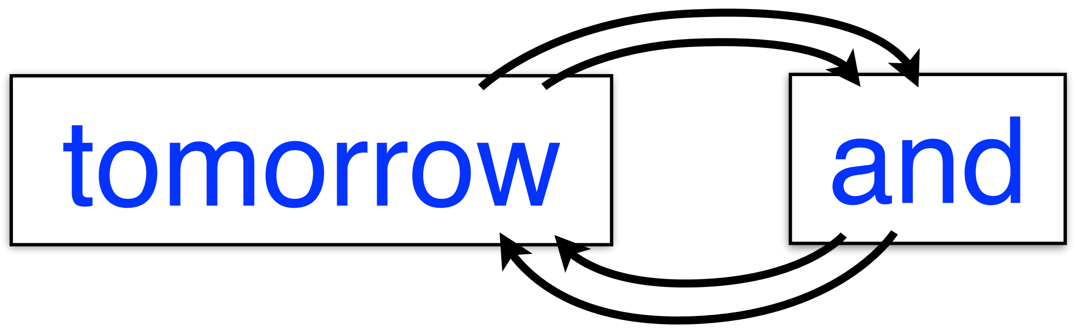
The de Bruijn Graph and its efficient representation

Different kind of graph

"tomorrow and tomorrow and tomorrow"



An edge represents an ordered pair of adjacent words in the input

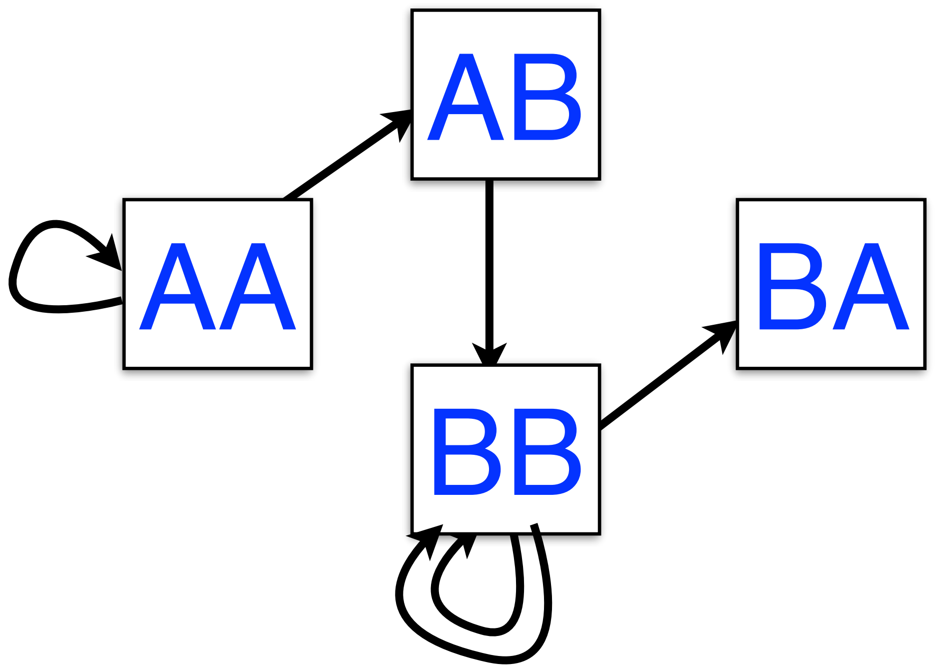
Multigraph: there can be more than one edge from node A to node B

De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB AB, BB BB, BB BB, BB BB, BA

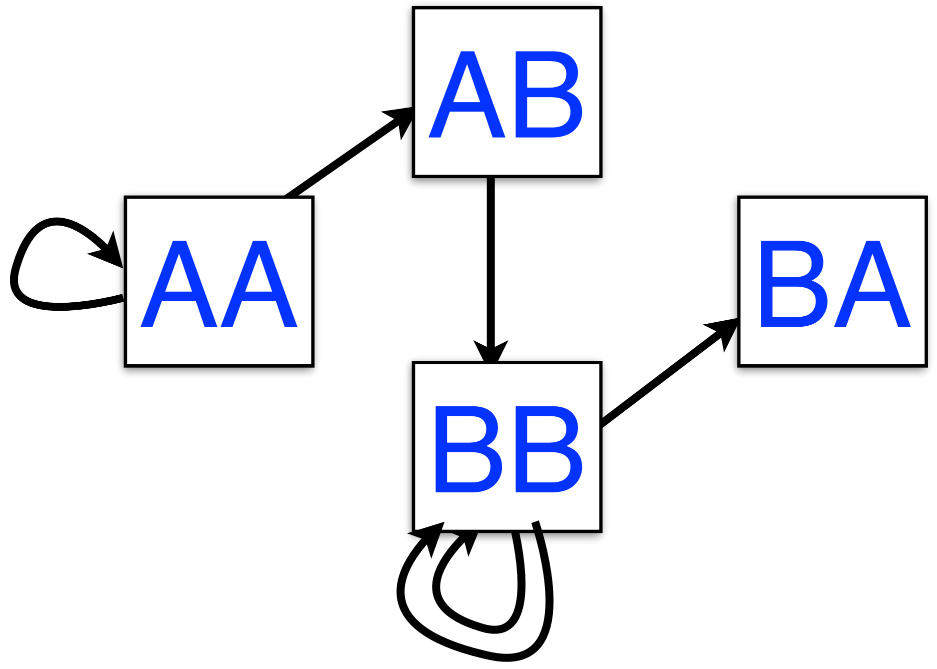


One edge per k-mer

One node per distinct k-1-mer

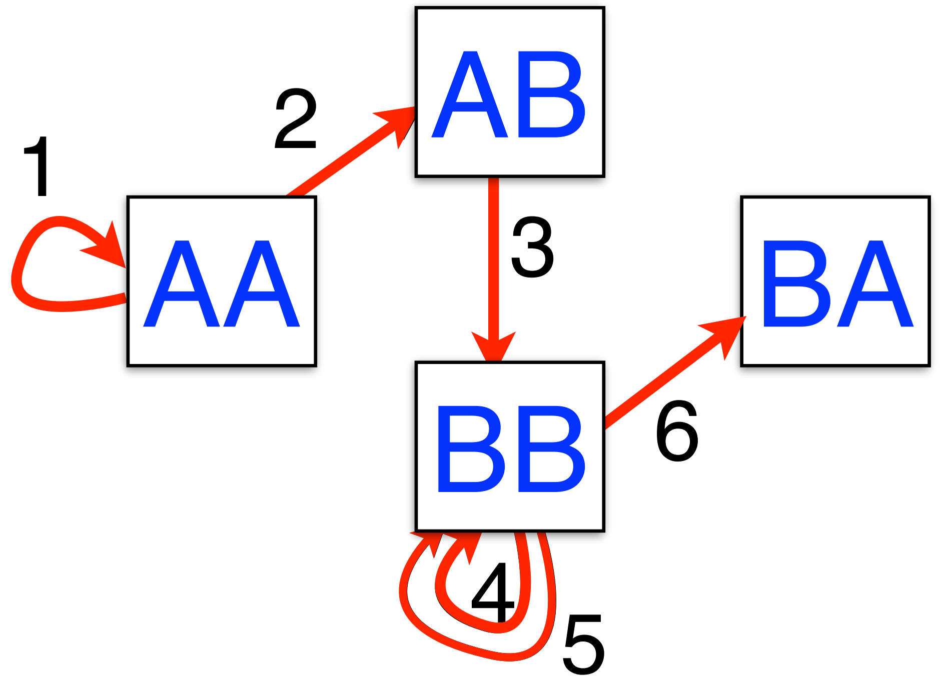
*

De Bruijn graph



Walk crossing each edge exactly once gives a reconstruction of the genome

De Bruijn graph



AAABBBBA

Walk crossing each edge exactly once gives a reconstruction of the genome . This is an Eulerian walk.

*

De Bruijn graph

Aside: how do you pronounce "De Bruijn"?

There is debate:

https://www.biostars.org/p/7186/

Nicolaas Govert de Bruijn
1918 -- 2012

*

Directed multigraph

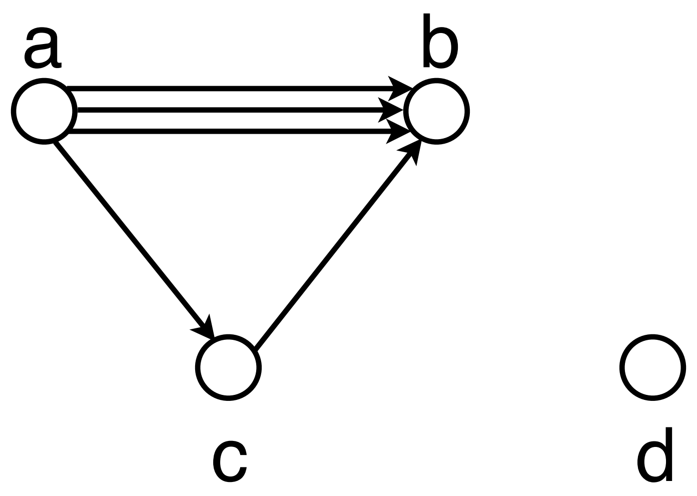
Directed multigraph G(V, E) consists of set of vertices, V and multiset of directed edges, E

Otherwise, like a directed graph

Node's indegree = # incoming edges

Node's outdegree = # outgoing edges

De Bruijn graph is a directed multigraph



$$V = \{ a, b, c, d \}$$

$$E = \{ \underbrace{(a, b), (a, b), (a, b)}_{\text{Repeated}}, (a, c), (c, b) \}$$

Eulerian walk definitions and statements

Node is balanced if indegree equals outdegree

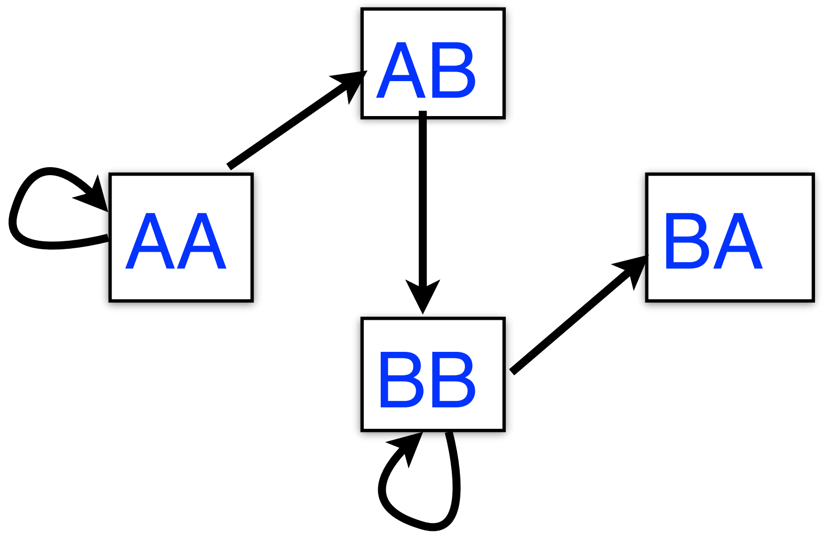
Node is semi-balanced if indegree differs from outdegree by 1

Graph is connected if each node can be reached by some other node

Eulerian walk visits each edge exactly once

Not all graphs have Eulerian walks. Graphs that do are Eulerian. (For simplicity, we won't distinguish Eulerian from semi-Eulerian.)

A directed, connected graph is Eulerian if and only if it has at most 2 semi-balanced nodes and all other nodes are balanced



Jones and Pevzner section 8.8

*

De Bruijn graph

Back to de Bruijn graph

$$\begin{tikzcd}
 & AB \arrow[dd] & \\
AA \arrow[loop left] \arrow[ur] & & BA \\
 & BB \arrow[loop below] \arrow[ur] &
\end{tikzcd}$$

$$\begin{tikzcd}
AAA \arrow[d] \arrow[dr] & & AAB \arrow[d] \arrow[dr] & & ABB \arrow[d] \arrow[dr] & & BBB \arrow[d] \arrow[dr] & & BBA \arrow[d] \arrow[dr] & \\
AA & AA & AA & AB & AB & BB & BB & BB & BB & BA \\
L & R & L & R & L & R & L & R & L & R
\end{tikzcd}$$

Is it Eulerian? Yes

Argument 1: AA → AA → AB → BB → BB → BA

Argument 2: AA and BA are semi-balanced, AB and BB are balanced

Bloom Filters & De Bruijn Graphs

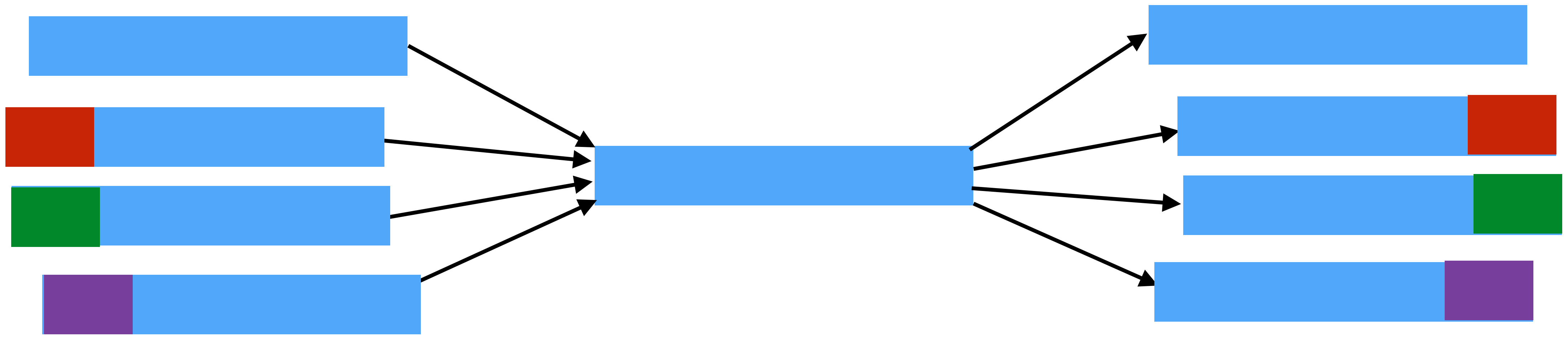
Recall the Bloom Filter: how could this data structure be useful for representing a De Bruijn graph?

Say we have a bloom filter B, for all of the k-mers in our data set, and say I give you one k-mer that is truly present.

We now have a "navigational" representation of the De Bruijn graph (can return the set of neighbors of a node, but not select/iterate over nodes); why?

Detour: Bloom Filters & De Bruijn Graphs

How could this data structure be useful for representing a De Bruijn graph?



A given (k-1)-mer can only have 2*|Σ| neighbors; |Σ| incoming and |Σ| outgoing neighbors — for genomes |Σ| = 4

To navigate in the De Bruijn graph, we can simply query all possible successors, and see which are actually present.

Bloom Filters & De Bruijn Graphs

But, a Bloom filter still has false-positives, right?

May return some neighbors that are not actually present.

Pell et al., PNAS 2012, use a lossy Bloom filter directly

Chikhi & Rizk, WABI 2012, present a *lossless* data structure based on Bloom filters

Salikhov et al., WABI 2013 extend this work and introduce the concept of “cascading” Bloom filters

Pellow, Filippova & Kingsford, RECOMB 2016. Take advantage of “independence” of false positives to lower FP rate for Bloom Filter representations

First, some bounds

JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 22, Number 5, 2015
© Mary Ann Liebert, Inc.
Pp. 336–352
DOI: 10.1089/cmb.2014.0160

Research Articles

On the Representation of De Bruijn Graphs

RAYAN CHIKHI,[1,6] ANTOINE LIMASSET,[3] SHAUN JACKMAN,[4]
JARED T. SIMPSON,[5] and PAUL MEDVEDEV[1,2,6]

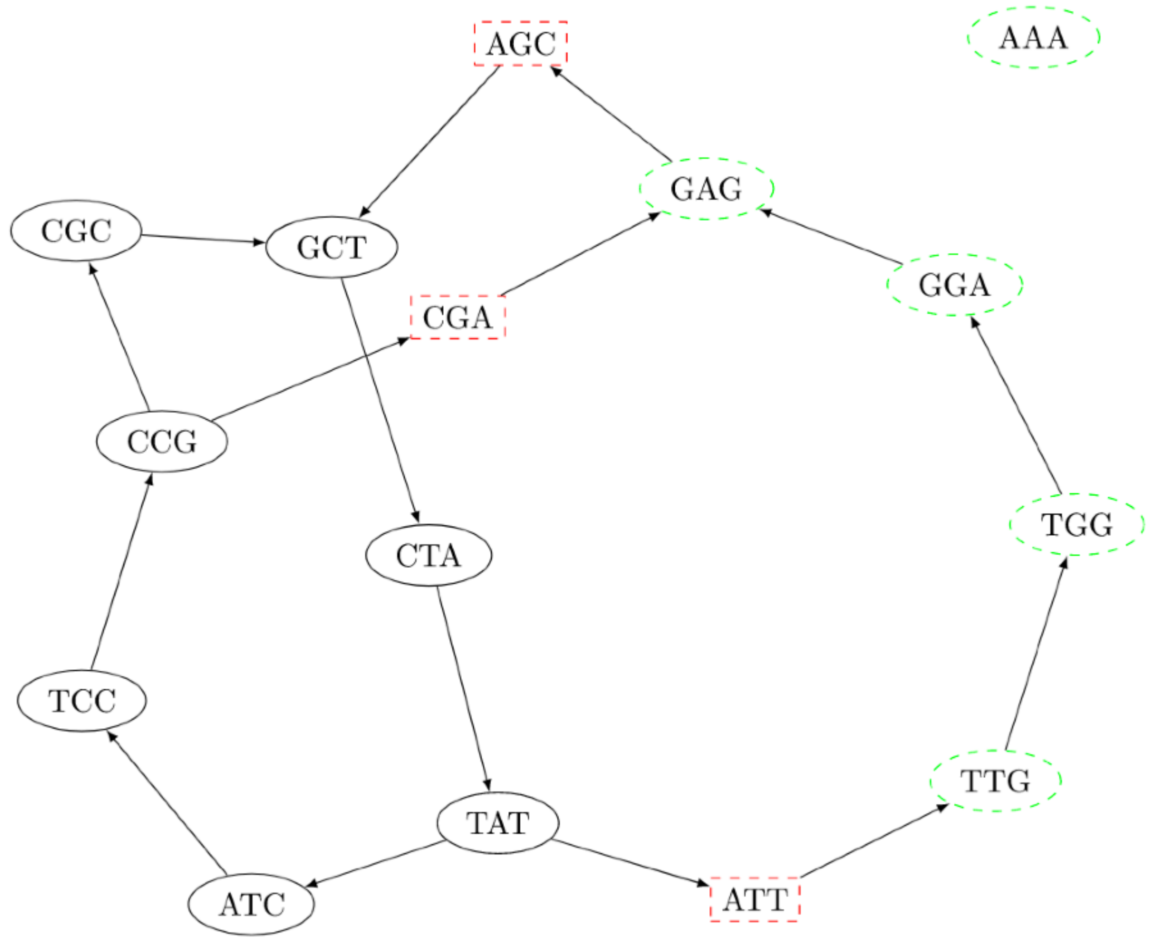
We use the term *membership data structure* to refer to a way of representing a dBG and answering k-mer membership queries. We can view this as a pair of algorithms: (CONST, MEMB). The CONST algorithm takes a set of k-mers S (i.e., a dBG) and outputs a bit string. We call CONST a constructor, since it constructs a representation of a dBG. The MEMB algorithm takes as input a bit string and a k-mer x and outputs true or false. Intuitively, MEMB takes a representation of a dBG created by CONST and outputs whether a given k-mer is present. Formally, we require that for all $x \in \Sigma^k$, MEMB(CONST(S), x) is true if and only if $x \in S$.

An NDS is a pair of algorithms, CONST and NBR. As before, CONST takes a set of k-mers and outputs a bit string. NBR takes a bit string and a k-mer and outputs a set of k-mers. The algorithms must satisfy that for every dBG S and a k-mer $x \in S$, NBR(CONST(S), x) $= ext(x) \cap S$. Note that if $x \notin S$, then the behavior of NBR(CONST(S), x) is undefined. We observe that a membership data structure immediately implies an NDS because an NBR query can be reduced to eight MEMB queries.

In this section, we prove that a navigational data structure on de Bruijn graphs needs at least 3.24 bits per k-mer to represent the graph:

Theorem 1. *Consider an arbitrary NDS and let* CONST *be its constructor. For any* $0 < \epsilon < 1$, *there exists a* k *and* $x \subseteq \Sigma^k$ *such that* $|\text{CONST}(x)| \geq |x| \cdot (c - \epsilon)$, *where* $c = 8 - 3\lg 3 \approx 3.25$.

Critical False Positives



(a)

$a_1 \ldots a_k$	$\sum_{i=1}^{k} a_i^i \bmod 10$
ATC	0
CCG	0
TCC	5
CGC	6
…	…

“toy” hash func (not used in practice)

(b)

Bloom filter
1
0
0
0
0
1
1
0
0
0

(c)

Nodes self-information:

$$\lceil log_2 \binom{4^3}{7} \rceil = 30 \text{ bits}$$

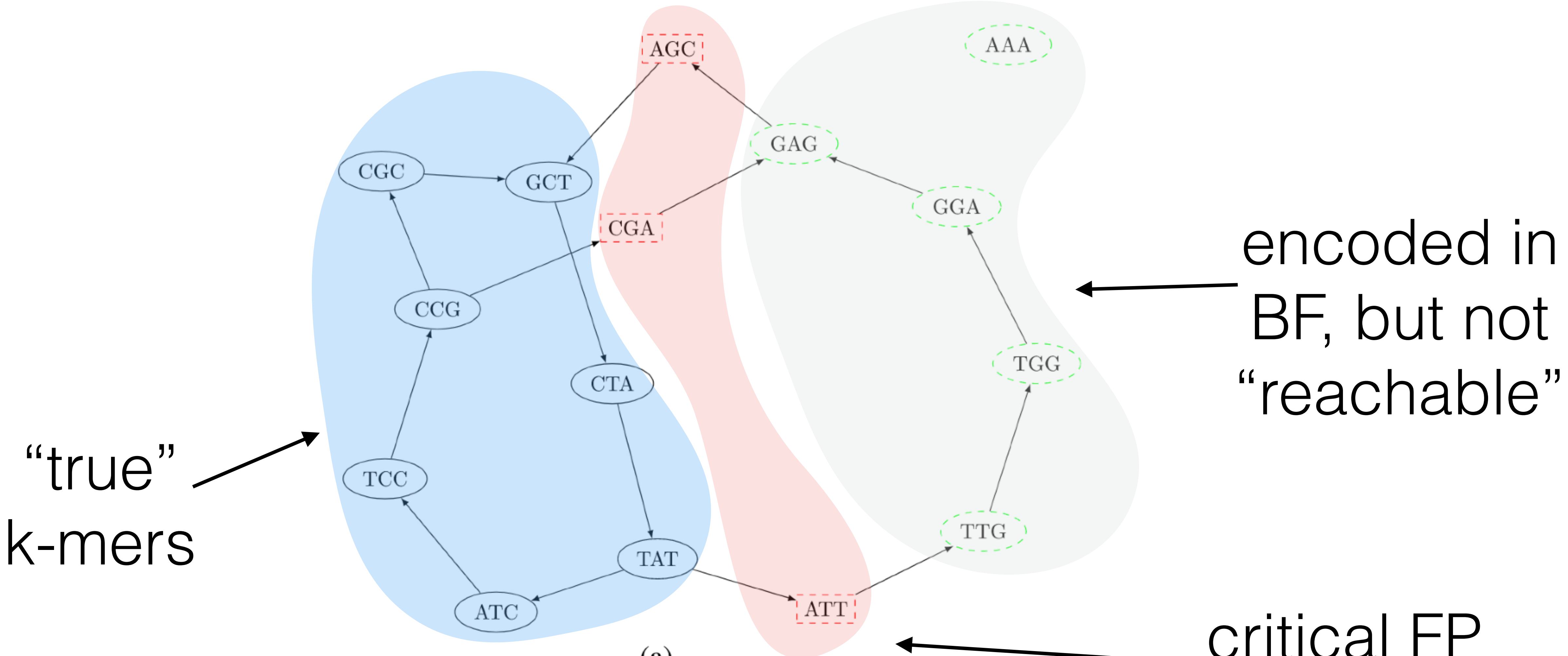
Structure size:

$$\underbrace{10}_{\text{Bloom}} + \underbrace{3 \cdot 6}_{\text{False positives}} = 28 \text{ bits}$$

(d)

Chikhi & Rizk

Critical False Positives



(a)

$a_1 \ldots a_k$	$\sum_{i=1}^{k} a_i^i \bmod 10$
ATC	0
CCG	0
TCC	5
CGC	6
...	...

"toy" hash func (not used in practice)

(b)

Bloom filter
1
0
0
0
0
1
1
0
0
0

(c)

Nodes self-information:

$$\lceil log_2 \binom{4^3}{7} \rceil = 30 \text{ bits}$$

Structure size:

$$\underbrace{10}_{\text{Bloom}} + \underbrace{3 \cdot 6}_{\text{False positives}} = 28 \text{ bits}$$

(d)

Chikhi & Rizk

Idea of Chkhi and Rizk

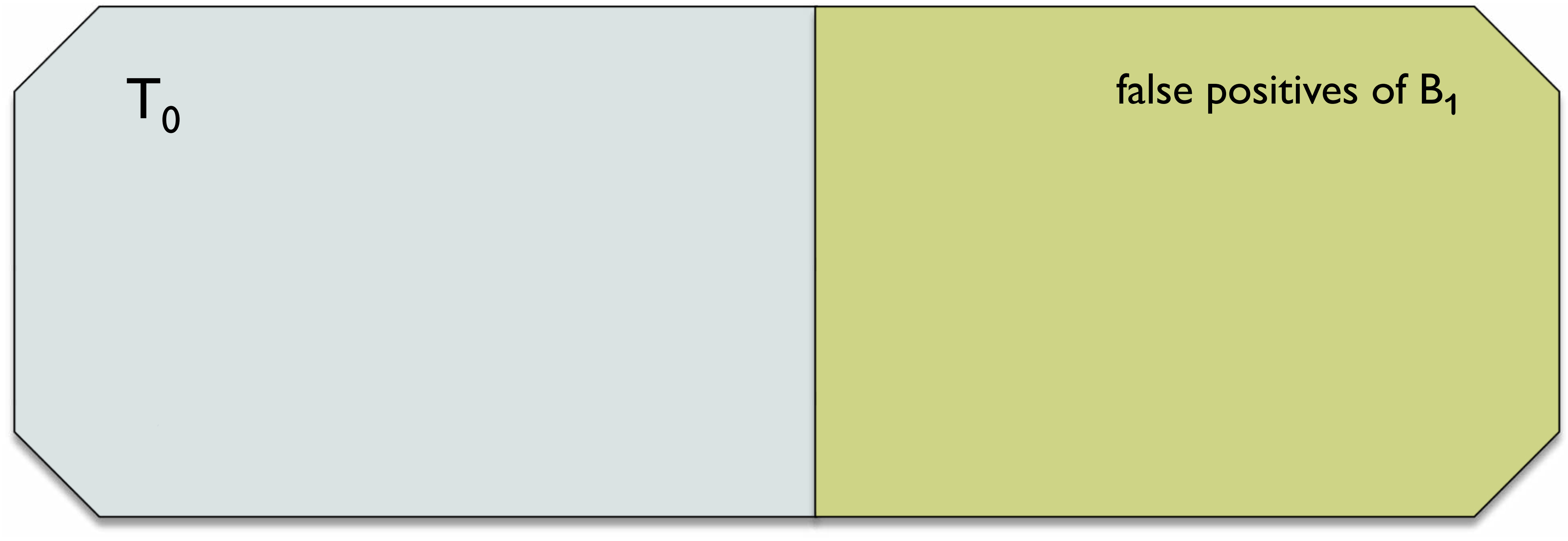
Assume we want to represent specific set T0 of k-mers with a Bloom filter B1

Key observation: in assembly, not all k-mers can be queried, only those having k-1 overlap with k-mers known to be in the graph.

The set T1 of "critical false positives" (false neighbors of true k-mers) is *much* smaller than the set of all false positives and can be stored explicitly

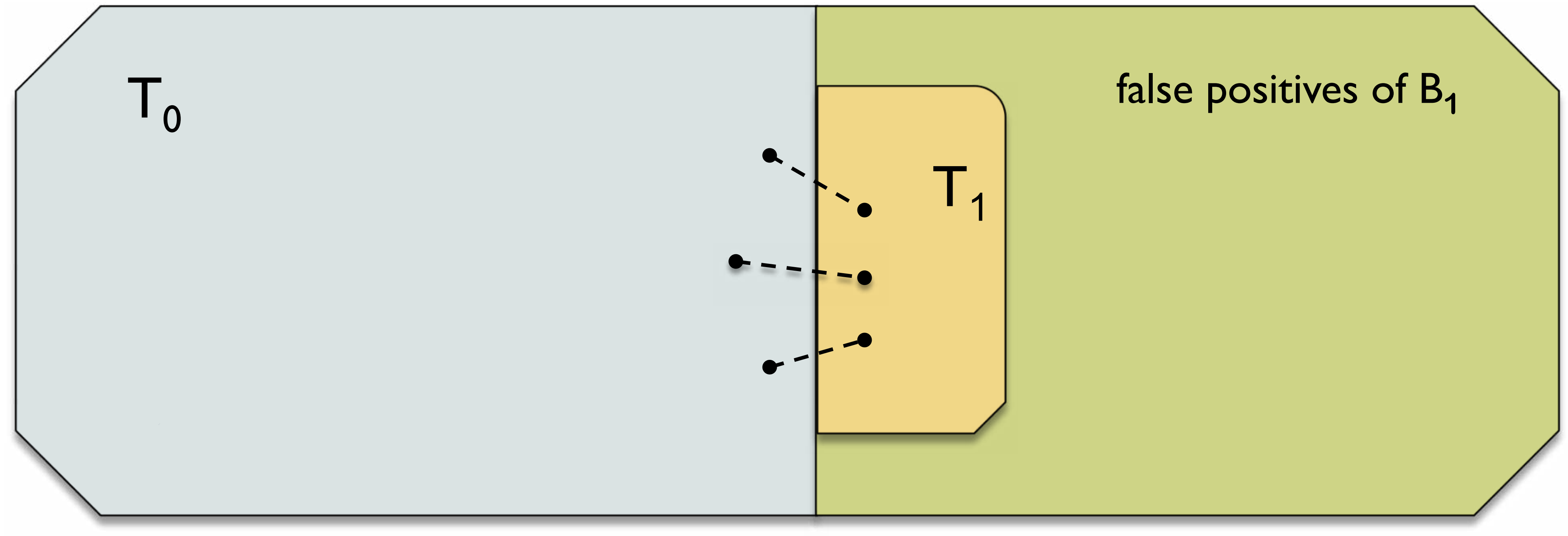
Storing B1 and T1 is much more space efficient that other exact methods for storing T0. Membership of w in T0 is tested by first querying B1, and if w ∊ B1, check that it is *not* in T1.

* slide courtesy of Salikhov, Sacomoto & Kucherov



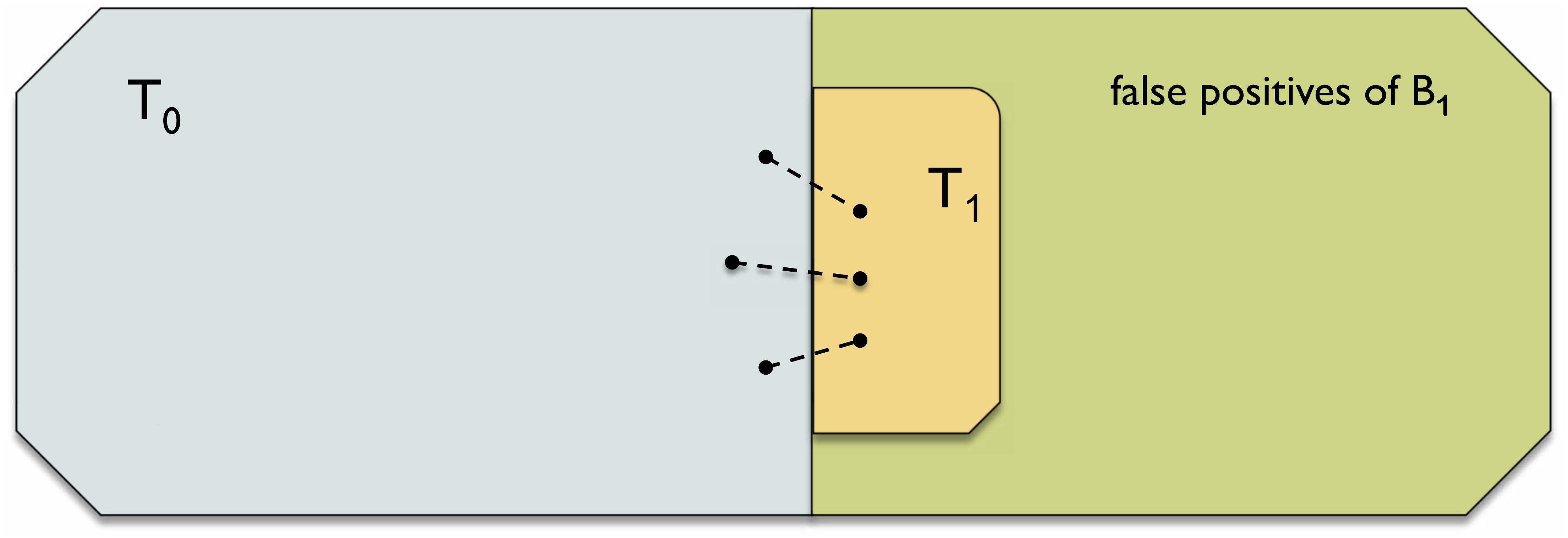
- Represent T_0 by Bloom filter B_1

* slide courtesy of Salikhov, Sacomoto & Kucherov



- Represent T_0 by Bloom filter B_1
- Compute T_1 ('critical false positives') and represent it e.g. by a hash table

* slide courtesy of Salikhov, Sacomoto & Kucherov



- Represent T_0 by Bloom filter B_1
- Compute T_1 ('critical false positives') and represent it e.g. by a hash table

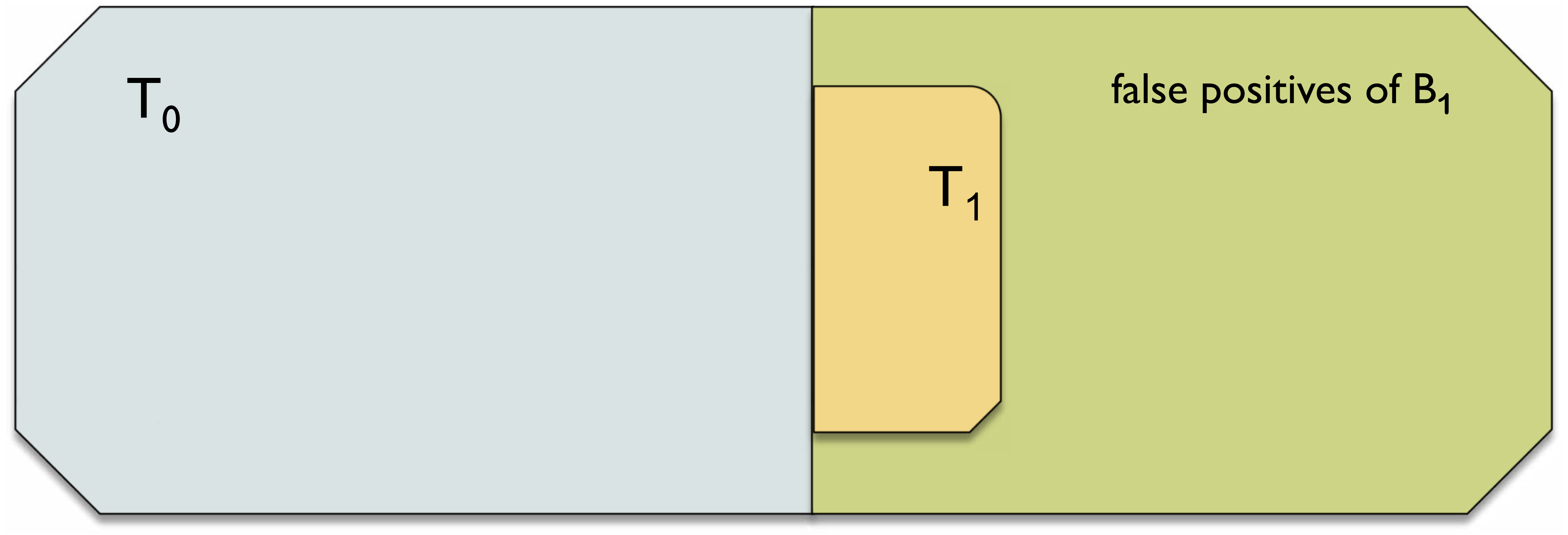
- Result (example): 13.2 bits/node for *k*=27 (of which 11.1 bits for B_1 and 2.1 bits for T_1)

* slide courtesy of Salikhov, Sacomoto & Kucherov

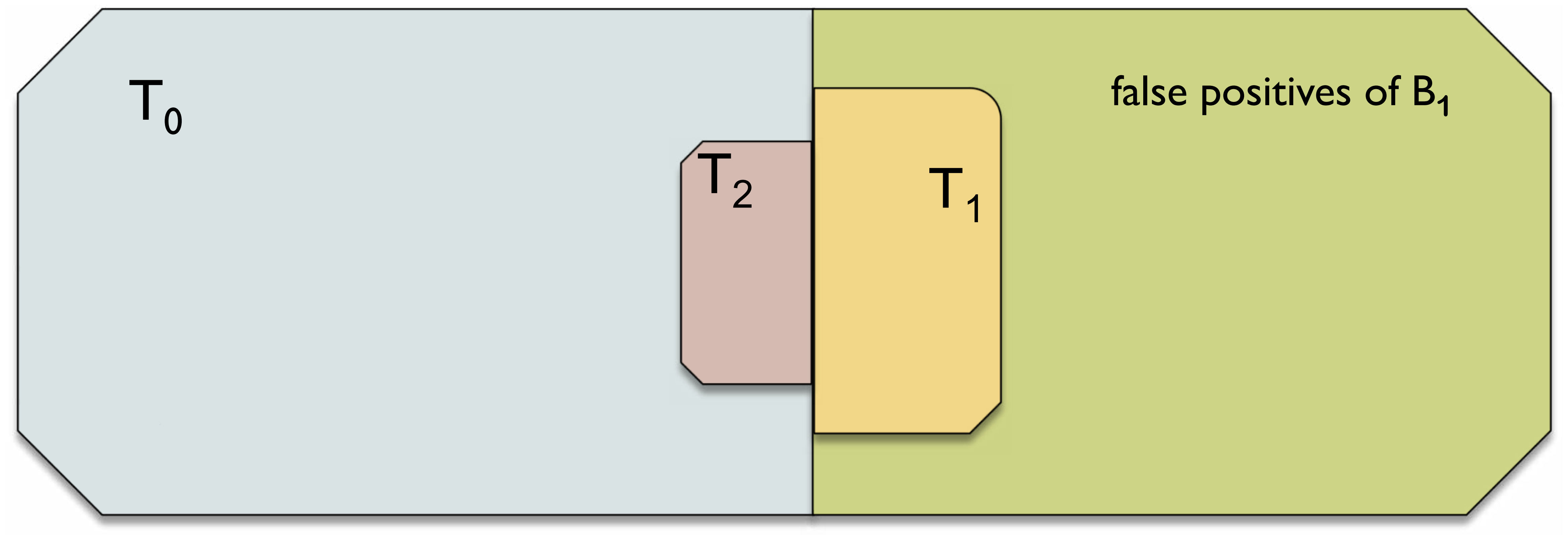
Improving on Chikhi and Rizk's method

- *Main idea*: iteratively apply the same construction to T_1 i.e. encode T_1 by a Bloom filter B_2 and set of 'false-false positives' T_2, then apply this to T_2 etc.
- ☞ *cascading Bloom filters*

* slide courtesy of Salikhov, Sacomoto & Kucherov

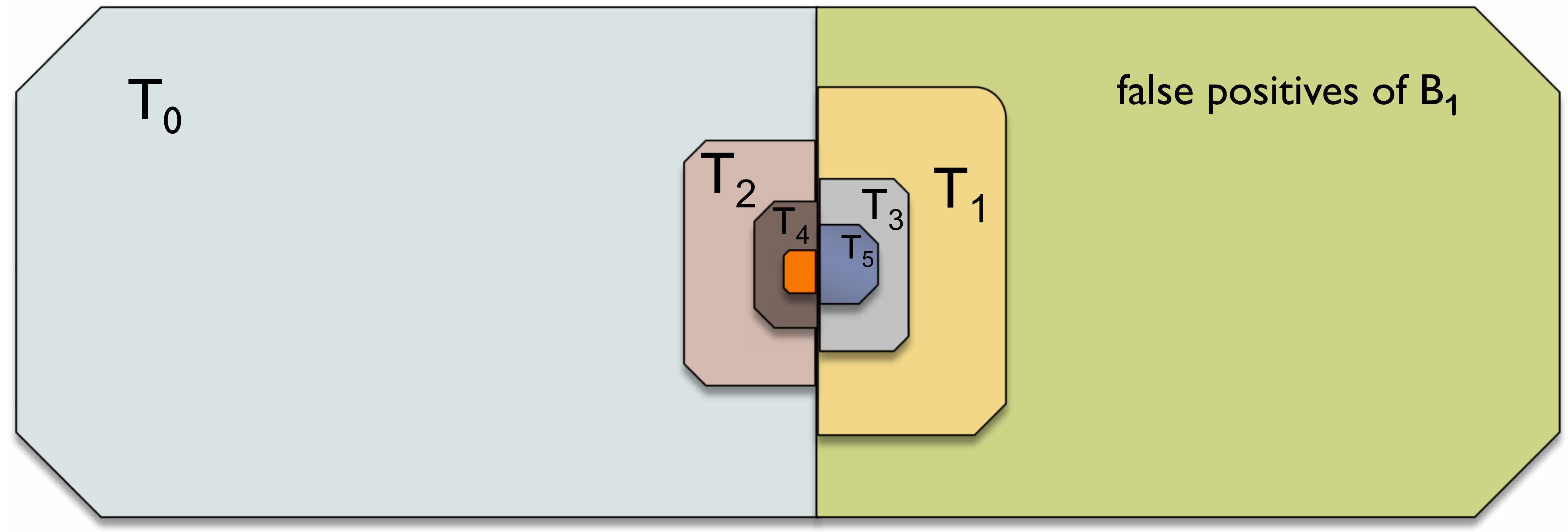


* slide courtesy of Salikhov, Sacomoto & Kucherov



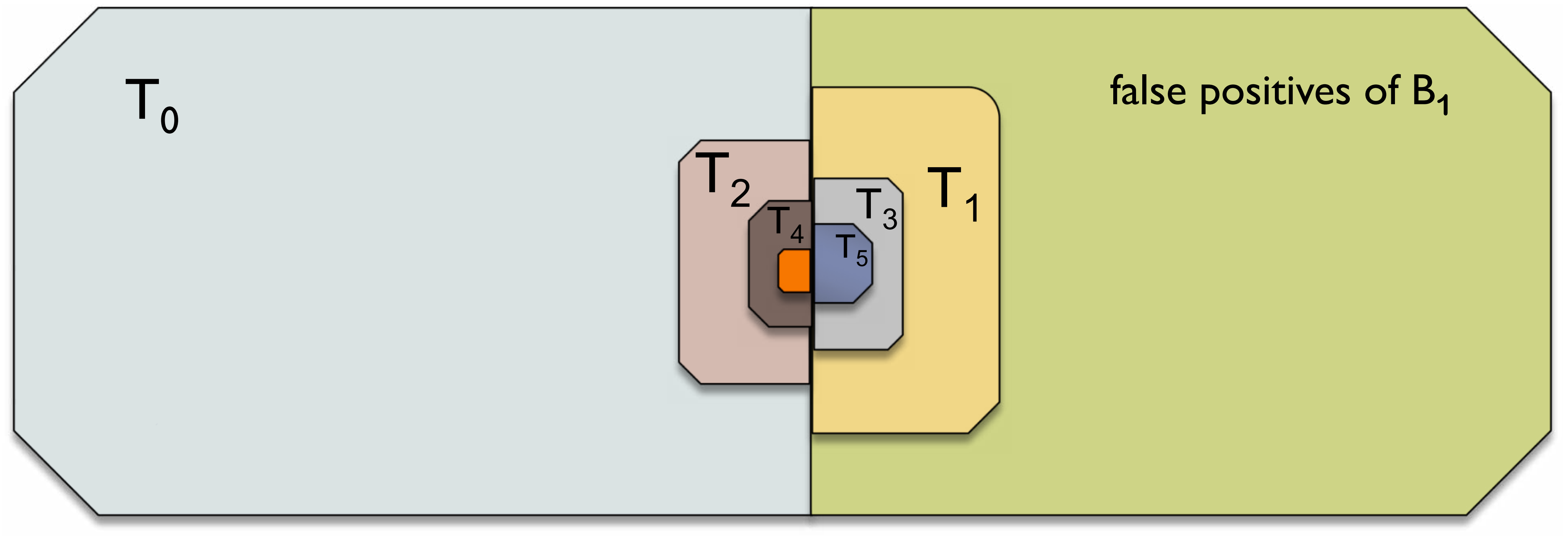
- further encode T_1 via a Bloom filter B_2 and set T_2, where $T_2 \subseteq T_0$ is the set of *k*-mers stored in B_2 by mistake ('false^2 positives')
-

* slide courtesy of Salikhov, Sacomoto & Kucherov



- further encode T_1 via a Bloom filter B_2 and set T_2, where $T_2 \subseteq T_0$ is the set of *k*-mers stored in B_2 by mistake ('false^2 positives')
- iterate the construction on T_2
- we obtain a sequence of sets $T_0, T_1, T_2, T_3, \ldots$ encode by Bloom filters $B_1, B_2, B_3, B_4, \ldots$ respectively
- $T_0 \supseteq T_2 \supseteq T_4 \supseteq \ldots, T_1 \supseteq T_3 \supseteq T_5 \supseteq$
-

* slide courtesy of Salikhov, Sacomoto & Kucherov



Lemma [correctness]: For a *k*-mer *w*, consider the smallest *i* such that $w \notin B_{i+1}$. Then $w \in T_0$ if *i* is odd and $w \notin T_0$ if *i* is even.

- if $w \notin B_1$ then $w \notin T_0$
- if $w \in B_1$, but $w \notin B_2$ then $w \in T_0$
- if $w \in B_1$, $w \in B_2$, but $w \notin B_3$ then $w \notin T_0$
- etc.

* slide courtesy of Salikhov, Sacomoto & Kucherov

Assuming infinite number of filters

Let N=|T$_0$| and $r=m_i/n_i$ is the same for every B$_i$. Then the total size is

$$\underbrace{rN}_{|B_1|} + \underbrace{6rNc^r}_{|B_2|} + \underbrace{rNc^r}_{|B_3|} + \underbrace{6rNc^{2r}}_{|B_4|} + \underbrace{rNc^{2r}}_{|B_5|} + \ldots = N(1+6c^r)\frac{r}{1-c^r}$$

The minimum is achieved for r=5.464, which yields the memory consumption of 8.45 bits/node

* slide courtesy of Salikhov, Sacomoto & Kucherov

Infinity difficult to deal with ;)

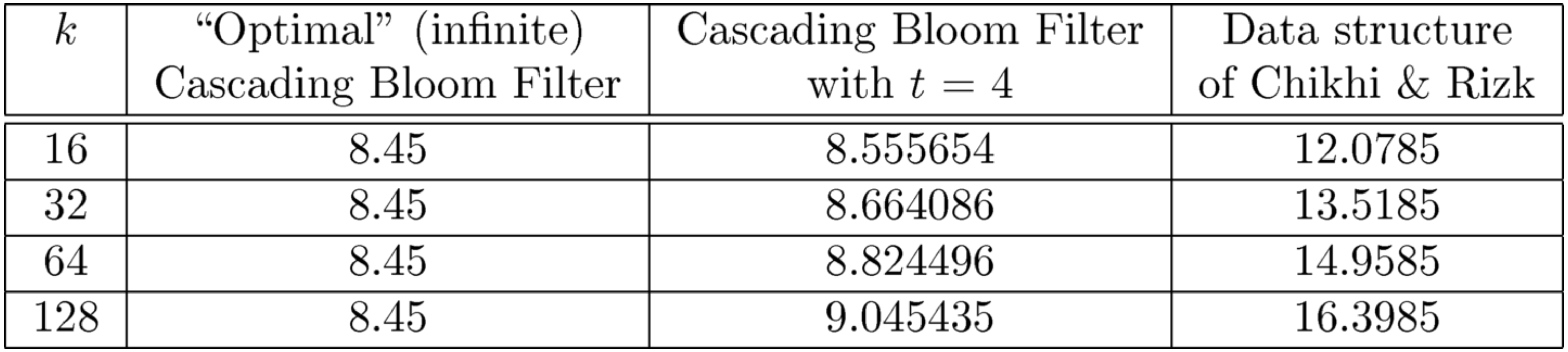
- In practice we will store only a small finite number of filters B_1, B_2,…, B_t together with the set T_t stored explicitely
- t=1 ➟ Chkhi&Rizk's method
- The estimation should be adjusted, optimal value of *r* has to be updated, example for t=4

k	optimal r	bits per k-mer
16	5.776737	8.555654
32	6.048557	8.664086
64	6.398529	8.824496
128	6.819496	9.045435

Table: Estimations for t=4. Optimal *r* and corresponding memory consumption

* slide courtesy of Salikhov, Sacomoto & Kucherov

Compared to Chikhi&Rizk's method



k	"Optimal" (infinite) Cascading Bloom Filter	Cascading Bloom Filter with $t = 4$	Data structure of Chikhi & Rizk
16	8.45	8.555654	12.0785
32	8.45	8.664086	13.5185
64	8.45	8.824496	14.9585
128	8.45	9.045435	16.3985

Table: Space (bits/node) compared to Chikhi&Rizk for *t*=4 and different values of *k*.

* slide courtesy of Salikhov, Sacomoto & Kucherov

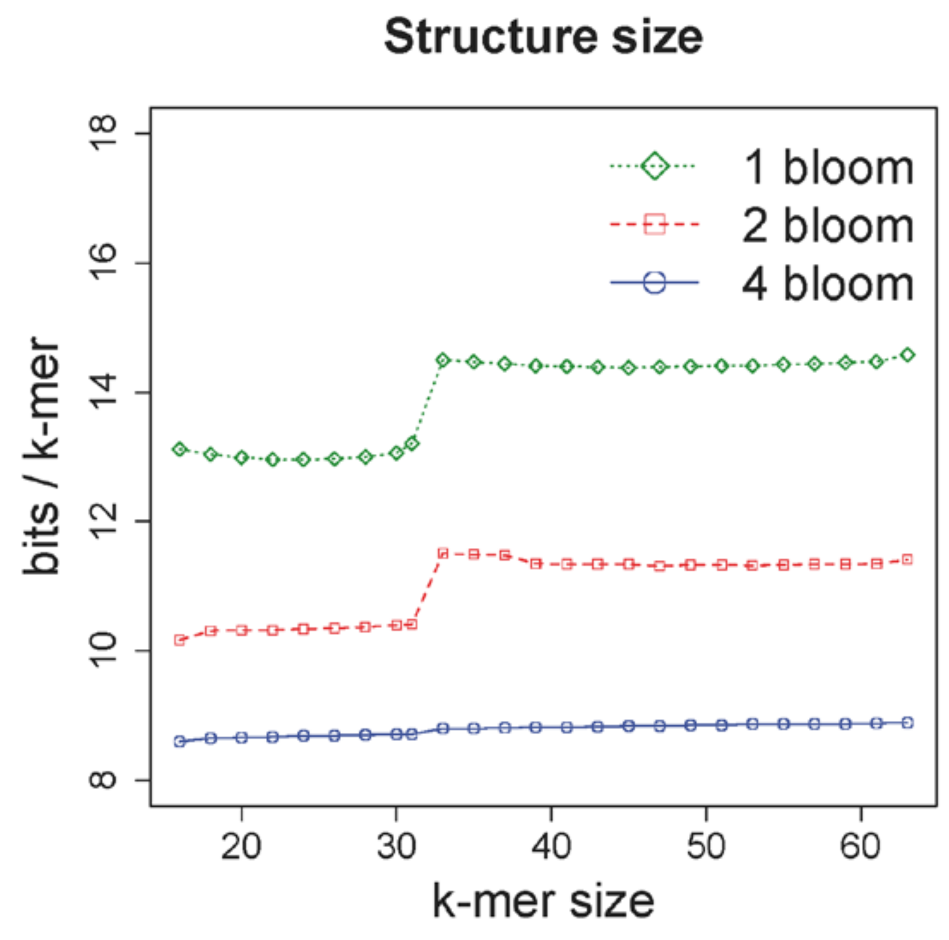
We can cut down a bit more …

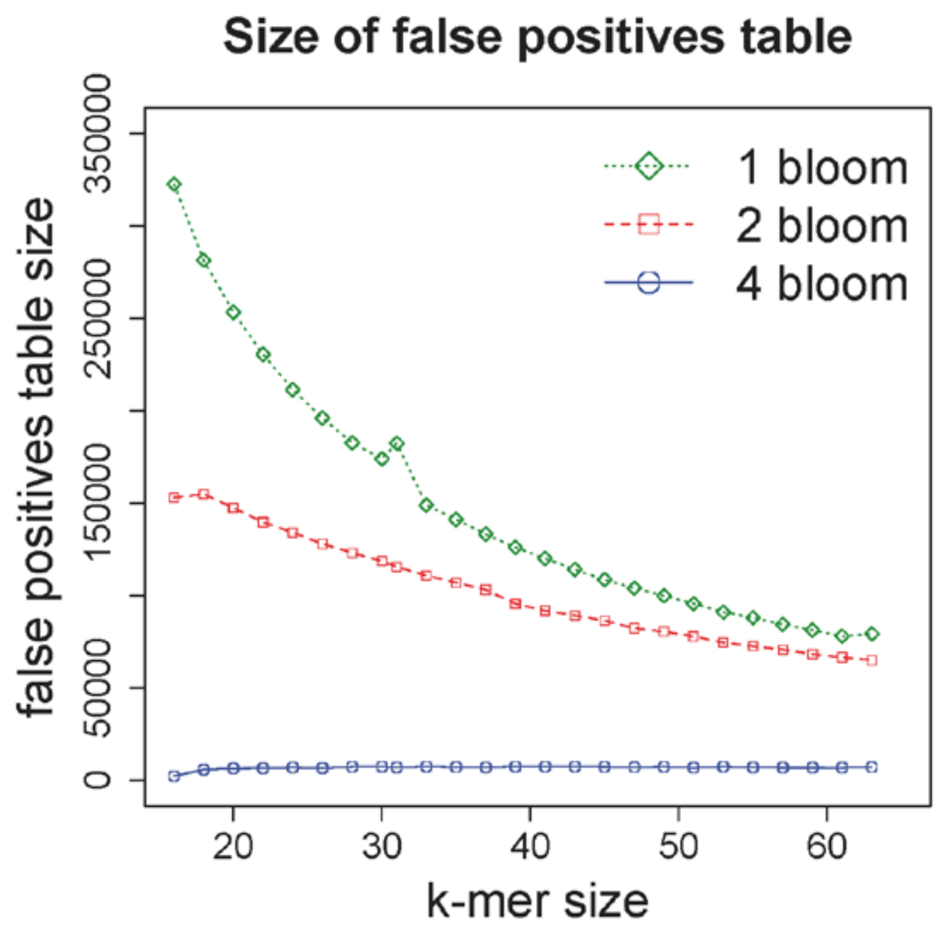
- Rather than using the same r for all filters $B_1, B_2, \ldots,$ we can use different properly chosen coefficients $r_1, r_2, \ldots$
- This allows saving another 0.2 – 0.4 bits/k-mer

* slide courtesy of Salikhov, Sacomoto & Kucherov

Experiments I: E.Coli, varying *k*

- 10M E.Coli reads of 100bp
- 3 versions compared: 1 Bloom (=Chikhi&Rizk), 2 Bloom (*t*=2) and 4 Bloom (*t*=4)

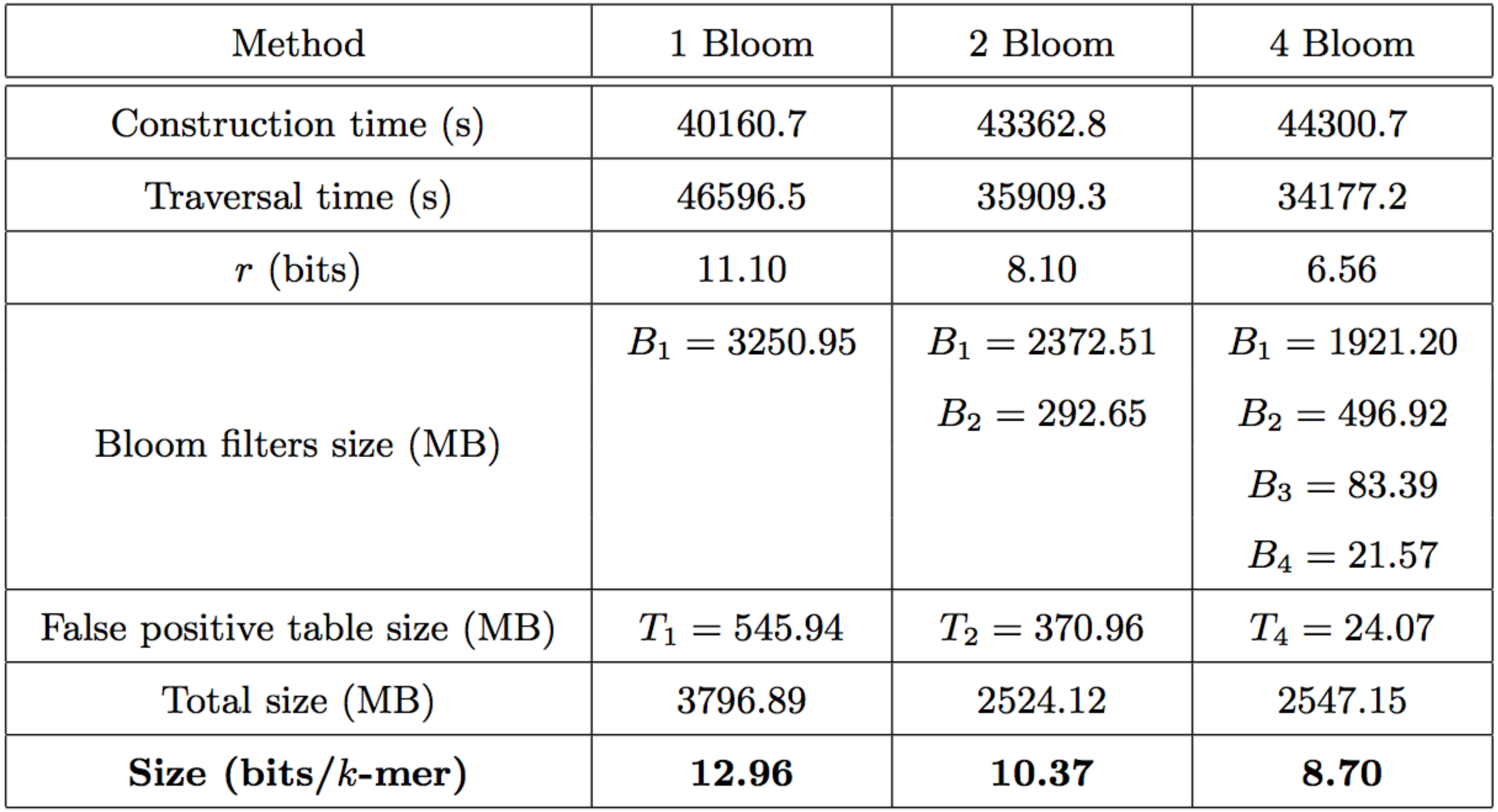




* slide courtesy of Salikhov, Sacomoto & Kucherov

Experiments II: Human dataset

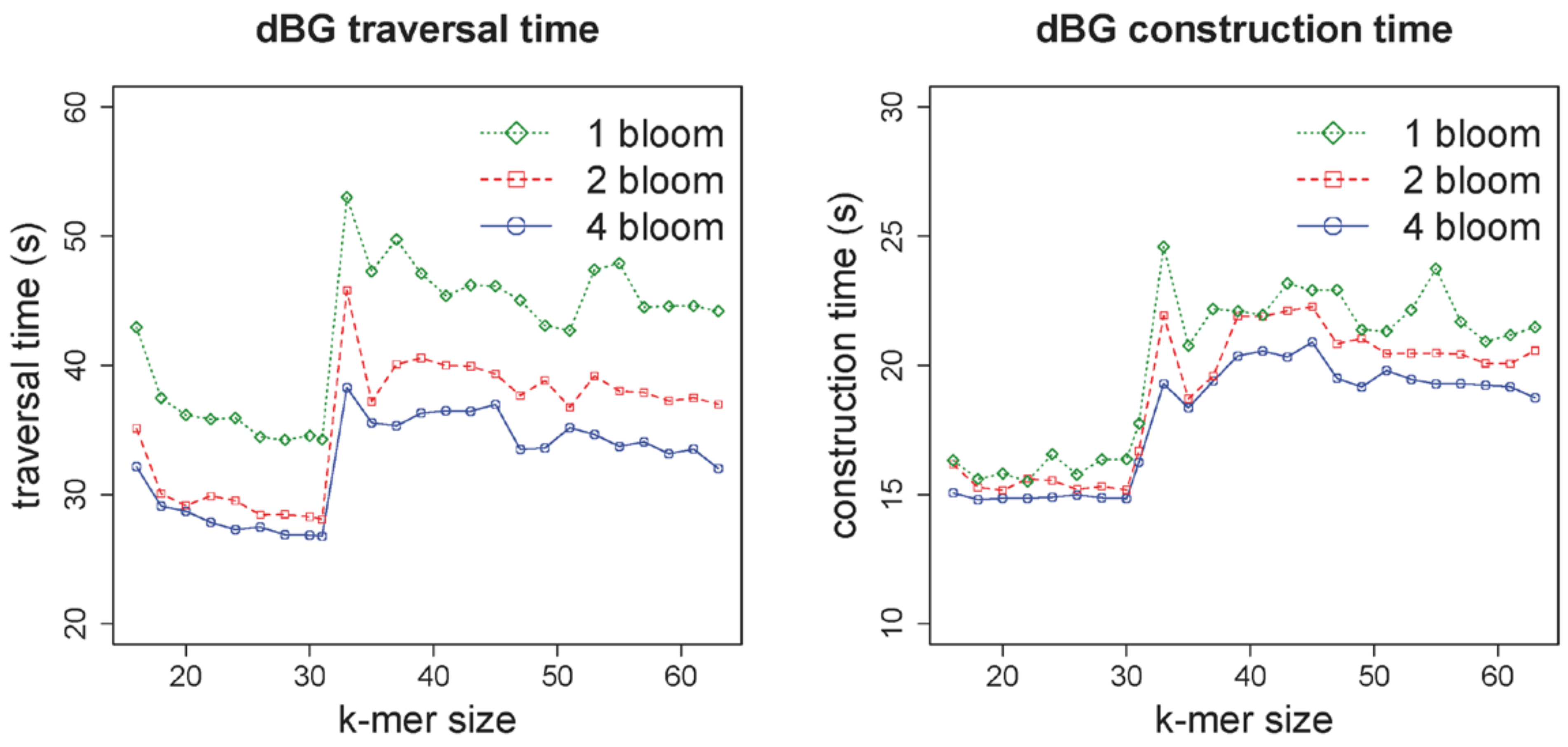
- 564M Human reads of 100bp (~17X coverage)



Method	1 Bloom	2 Bloom	4 Bloom
Construction time (s)	40160.7	43362.8	44300.7
Traversal time (s)	46596.5	35909.3	34177.2
r (bits)	11.10	8.10	6.56
Bloom filters size (MB)	$B_1 = 3250.95$	$B_1 = 2372.51$ $B_2 = 292.65$	$B_1 = 1921.20$ $B_2 = 496.92$ $B_3 = 83.39$ $B_4 = 21.57$
False positive table size (MB)	$T_1 = 545.94$	$T_2 = 370.96$	$T_4 = 24.07$
Total size (MB)	3796.89	2524.12	2547.15
Size (bits/k-mer)	**12.96**	**10.37**	**8.70**

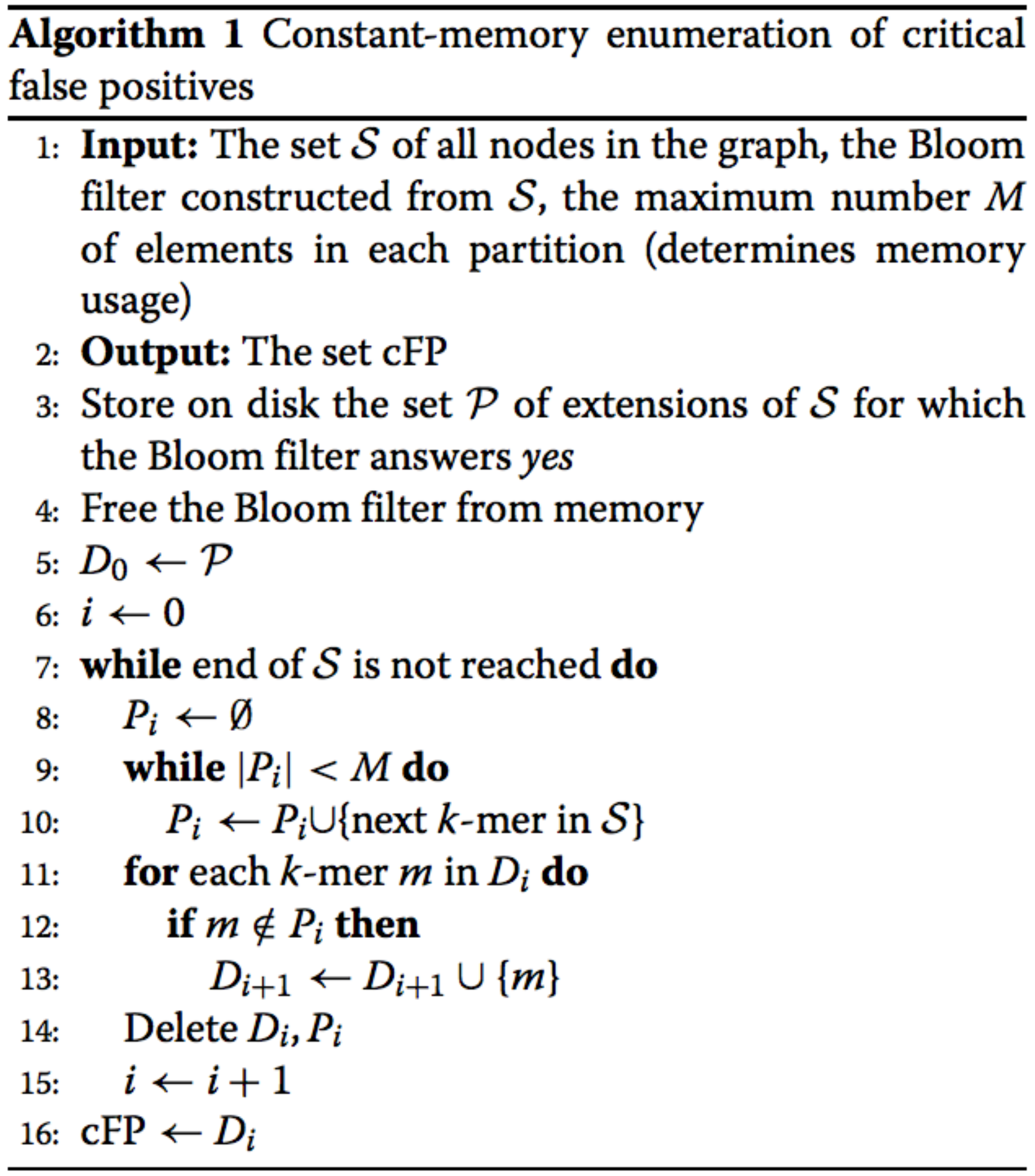
* slide courtesy of Salikhov, Sacomoto & Kucherov

Experiments I (cont)



* slide courtesy of Salikhov, Sacomoto & Kucherov

Efficiently enumerating cFP

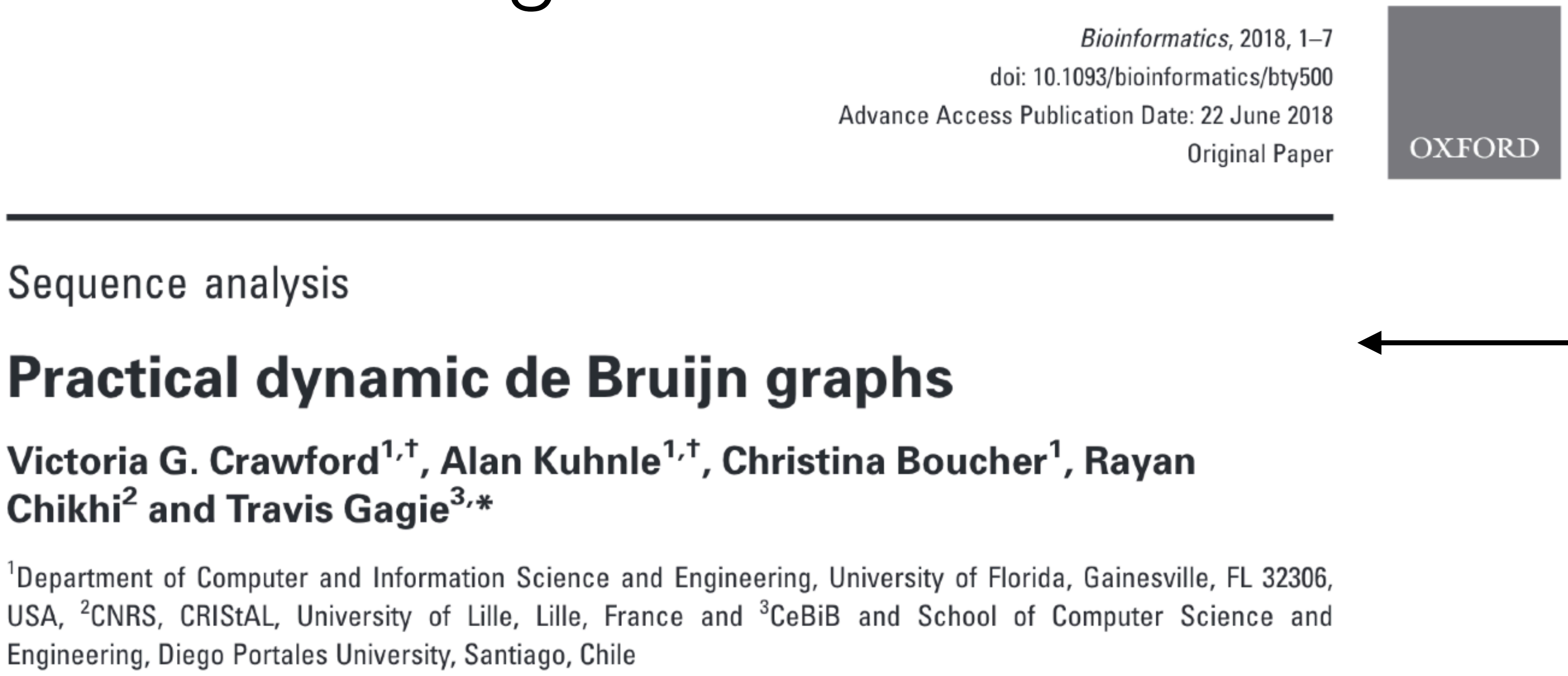


Algorithm 1 Constant-memory enumeration of critical false positives

```
1:  Input: The set $\mathcal{S}$ of all nodes in the graph, the Bloom filter constructed from $\mathcal{S}$, the maximum number $M$ of elements in each partition (determines memory usage)
2:  Output: The set cFP
3:  Store on disk the set $\mathcal{P}$ of extensions of $\mathcal{S}$ for which the Bloom filter answers yes
4:  Free the Bloom filter from memory
5:  $D_0 \leftarrow \mathcal{P}$
6:  $i \leftarrow 0$
7:  while end of $\mathcal{S}$ is not reached do
8:      $P_i \leftarrow \emptyset$
9:      while $|P_i| < M$ do
10:         $P_i \leftarrow P_i \cup \{$next $k$-mer in $\mathcal{S}\}$
11:     for each $k$-mer $m$ in $D_i$ do
12:         if $m \notin P_i$ then
13:             $D_{i+1} \leftarrow D_{i+1} \cup \{m\}$
14:     Delete $D_i, P_i$
15:     $i \leftarrow i + 1$
16: cFP $\leftarrow D_i$
```

Bloom filters & De Bruijn Graphs

So, we can make very small representation of the dBG.
But it's navigational! We can also make them:



Bioinformatics, 2018, 1–7
doi: 10.1093/bioinformatics/bty500
Advance Access Publication Date: 22 June 2018
Original Paper

OXFORD

Sequence analysis

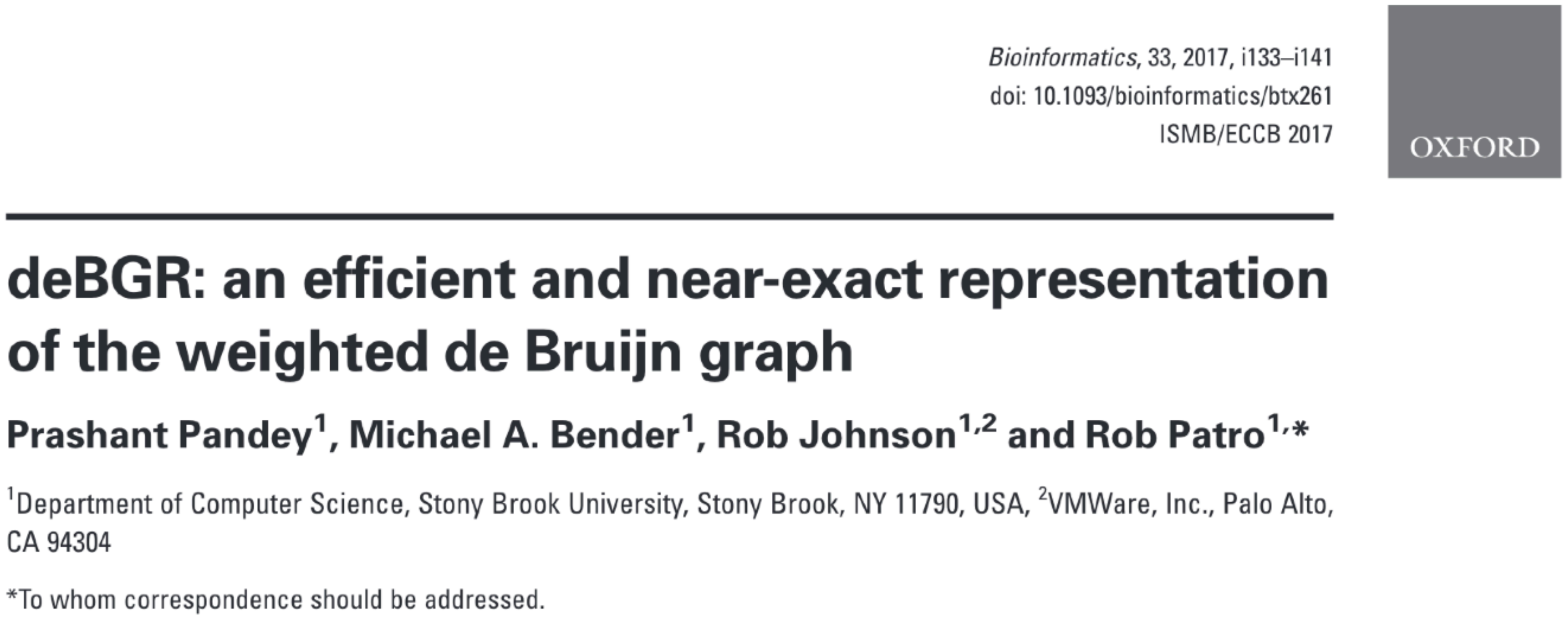
Practical dynamic de Bruijn graphs

Victoria G. Crawford[1,†], Alan Kuhnle[1,†], Christina Boucher[1], Rayan Chikhi[2] and Travis Gagie[3,*]

[1]Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32306, USA, [2]CNRS, CRIStAL, University of Lille, Lille, France and [3]CeBiB and School of Computer Science and Engineering, Diego Portales University, Santiago, Chile

$\longleftarrow$ Dynamic & membership

and even weighted $\longrightarrow$



Bioinformatics, 33, 2017, i133–i141
doi: 10.1093/bioinformatics/btx261
ISMB/ECCB 2017

OXFORD

deBGR: an efficient and near-exact representation of the weighted de Bruijn graph

Prashant Pandey[1], Michael A. Bender[1], Rob Johnson[1,2] and Rob Patro[1,*]

[1]Department of Computer Science, Stony Brook University, Stony Brook, NY 11790, USA, [2]VMWare, Inc., Palo Alto, CA 94304

*To whom correspondence should be addressed.