A BWT-based algorithm for random de Bruijn sequence construction

Zsuzsanna Lipták[1]

(joint work with Luca Parmigiani[2])

[1]University of Verona, Italy
[2]Bielefeld University, Germany

DCC, University of Chile, Santiago
27 March 2024

Def. A binary de Bruijn sequence (dB sequence) of order k is a (circular) string in which every k-mer (string of length k) occurs exactly once as a substring.

Ex. $t = \underset{0\,1\,2\,3\,4\,5\,6\,7}{\text{aaababbb}}$

de Bruijn sequences

Def. A binary de Bruijn sequence (dB sequence) of order k is a (circular) string in which every k-mer (string of length k) occurs exactly once as a substring.

Ex. $t = \underset{0\,1\,2\,3\,4\,5\,6\,7}{\text{aaababbb}}$

k-mer	position
aaa	0
aab	1
aba	2
abb	4
baa	7
bab	3
bba	6
bbb	5

de Bruijn sequences

Def. A binary de Bruijn sequence (dB sequence) of order k is a (circular) string in which every k-mer (string of length k) occurs exactly once as a substring.

Ex. $t = \underset{0\,1\,2\,3\,4\,5\,6\,7}{\text{aaababbb}}$

k-mer	position
aaa	0
aab	1
aba	2
abb	4
baa	7
bab	3
bba	6
bbb	5

Clearly, a dB sequence of order k has length 2^k.

de Bruijn sequences

- de Bruijn sequences exist for every k (Fly Sainte-Marie, 1894)
- There are $2^{2^{k-1}-k}$ dB sequences of order k (de Bruijn, 1946)

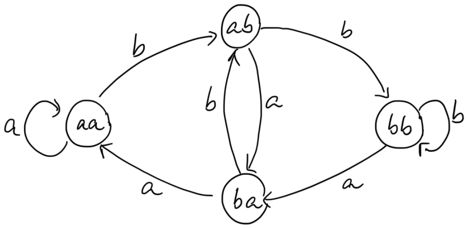
k	1	2	3	4	5	6	7	10	15
#dBseqs	1	1	2	16	2048	67 108 864	$1.44 \cdot 10^{17}$	$1.3 \cdot 10^{151}$	$3.63 \cdot 10^{4927}$

- $k = 1$: ab, $k = 2$: aabb, $k = 3$: aaababbb, aaabbbab
- dB sequences correspond to Euler cycles in the dB graph

de Bruijn graphs

Def. The (binary) de Bruijn graph of order k is a directed graph (V, E) s.t. $V = \{a, b\}^k$, and $(u, v) \in E$ iff there is $w \in \{a, b\}^{k+1}$ with prefix u and suffix v.[1]

Ex. $k = 2$:



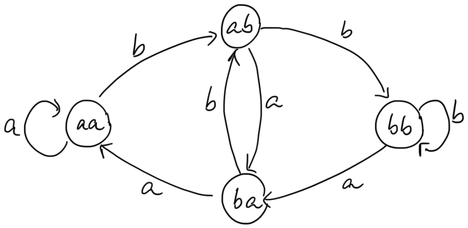
We write the new character x on edge (u, v): $w = ux$.

[1] In the bioinformatics literature these are called dB graphs of order $k + 1$.

de Bruijn graphs

Def. The (binary) de Bruijn graph of order k is a directed graph (V, E) s.t. $V = \{a, b\}^k$, and $(u, v) \in E$ iff there is $w \in \{a, b\}^{k+1}$ with prefix u and suffix v.[1]

Ex. $k = 2$:



We write the new character x on edge (u, v): $w = ux$.

So we have a 1-to-1 correspondence between E and $\{a, b\}^{k+1}$, and every walk in the dB graph spells a string (concatenate the new characters).

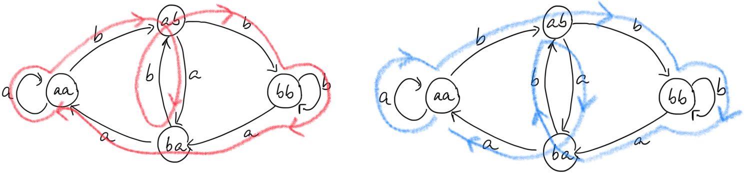
[1] In the bioinformatics literature these are called dB graphs of order $k + 1$.

de Bruijn graphs

- de Bruijn graphs are connected and balanced (all v: indeg = outdeg)
- By Euler's theorem, they are Eulerian (have Euler cycles).
- dB sequences of order k = Euler cycles in dB graph of order $k-1$

de Bruijn graphs

- de Bruijn graphs are connected and balanced (all v: indeg $=$ outdeg)
- By Euler's theorem, they are Eulerian (have Euler cycles).
- dB sequences of order k = Euler cycles in dB graph of order $k-1$

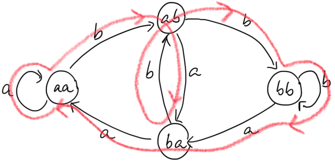


aaababbb

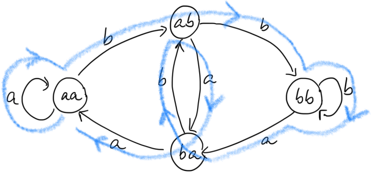
aaabbbab

de Bruijn graphs

- de Bruijn graphs are connected and balanced (all v: indeg $=$ outdeg)
- By Euler's theorem, they are Eulerian (have Euler cycles).
- dB sequences of order k = Euler cycles in dB graph of order $k - 1$



aaababbb



aaabbbab

- Tatyana Ehrenfest and Nicolaas de Bruijn gave the exact number of Euler cycles in directed Eulerian graphs (BEST theorem, 1951).

Applications of de Bruijn sequences

- pseudo-random bit generators

Applications of de Bruijn sequences

- pseudo-random bit generators
- experimental design: reaction time experiments, imaging studies (MRI)

Applications of de Bruijn sequences

- pseudo-random bit generators
- experimental design: reaction time experiments, imaging studies (MRI)
- computational biology: DNA probe design, DNA microarray, DNA synthesis

Applications of de Bruijn sequences

- pseudo-random bit generators
- experimental design: reaction time experiments, imaging studies (MRI)
- computational biology: DNA probe design, DNA microarray, DNA synthesis
- cryptography

Related work

Many algorithms exist for constructing dB sequences (see the classic book [Golomb 1968], the survey [Fredricksen 1982], Joe Sawada's website `debruijnsequence.org`). Most construct:

Related work

Many algorithms exist for constructing dB sequences (see the classic book [Golomb 1968], the survey [Fredricksen 1982], Joe Sawada's website `debruijnsequence.org`). Most construct:

- one particular dB sequence (e.g. the lex-least dB sequence), or

Related work

Many algorithms exist for constructing dB sequences (see the classic book [Golomb 1968], the survey [Fredricksen 1982], Joe Sawada's website `debruijnsequence.org`). Most construct:

- one particular dB sequence (e.g. the lex-least dB sequence), or
- a small subset of dB sequences (e.g. LFSRs = linear feedback shift registers)

Related work

Many algorithms exist for constructing dB sequences (see the classic book [Golomb 1968], the survey [Fredricksen 1982], Joe Sawada's website `debruijnsequence.org`). Most construct:

- one particular dB sequence (e.g. the lex-least dB sequence), or
- a small subset of dB sequences (e.g. LFSRs = linear feedback shift registers)

k	4	5	6	7	10	15	20
#LFSRs	2	6	6	18	60	1 800	24 000
#dBseqs	16	2048	67 108 864	$1.44 \cdot 10^{17}$	$1.3 \cdot 10^{151}$	$3.63 \cdot 10^{4927}$	$2.47 \cdot 10^{157820}$

Related work

Many algorithms exist for constructing dB sequences (see the classic book [Golomb 1968], the survey [Fredricksen 1982], Joe Sawada's website `debruijnsequence.org`). Most construct:

- one particular dB sequence (e.g. the lex-least dB sequence), or
- a small subset of dB sequences (e.g. LFSRs = linear feedback shift registers)

k	4	5	6	7	10	15	20
#LFSRs	2	6	6	18	60	1 800	24 000
#dBseqs	16	2048	67 108 864	$1.44 \cdot 10^{17}$	$1.3 \cdot 10^{151}$	$3.63 \cdot 10^{4927}$	$2.47 \cdot 10^{157820}$

- The only algorithms able to construct **any** dB sequence are based on finding Eulerian cycles in de Bruijn graphs (Hierholzer, Fleury)

Construction of random dB sequences

- Surprisingly, there appear to be no practical algorithm for random dB sequence construction that can output **any** dB sequence with positive probability.
- Our algorithm does just that!

The Burrows-Wheeler Transform

Def. The Burrows-Wheeler Transform (BWT) of a string t is the concatenation of the last characters of its rotations, taken in lexicographical order.

Ex. $t = \text{aaababbb}$

							bwt(t)
a	a	a	b	a	b	b	b
a	a	b	a	b	b	b	a
a	b	a	b	b	b	a	a
a	b	b	b	a	a	a	b
b	a	a	a	b	a	b	b
b	a	b	b	b	a	a	a
b	b	a	a	a	b	a	b
b	b	b	a	a	a	b	a

The Burrows-Wheeler Transform

Def. The Burrows-Wheeler Transform (BWT) of a string t is the concatenation of the last characters of its rotations, taken in lexicographical order.

Ex. $t =$ aaababbb

							bwt(t)
a	a	a	b	a	b	b	b
a	a	b	a	b	b	b	a
a	b	a	b	b	b	a	a
a	b	b	b	a	a	a	b
b	a	a	a	b	a	b	b
b	a	b	b	b	a	a	a
b	b	a	a	a	b	a	b
b	b	b	a	a	a	b	a

bwt(aaababbb) $=$ baabbaba

Reversing the BWT

Def. Given a string v, its standard permutation π_v is defined by: $\pi_v(i) < \pi_v(j)$ if (i) $v_i < v_j$, or (ii) $v_i = v_j$ and $i < j$.

(When v is a BWT, then π_v is also called LF-mapping, which can be used to recover t from $\text{bwt}(t)$ back-to-front.)

Ex. $v = \text{baabbaba}$

$$\pi_v = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 0 & 1 & 5 & 6 & 2 & 7 & 4 \end{smallmatrix}\right) = (0, 4, 6, 7, 3, 5, 2, 1)$$

Thm. (Folklore) A string v is the BWT of a primitive string u if and only if π_v is cyclic.

The BWT of a dB sequence

$t =$ aaababbb

							bwt(t)
a	a	a	b	a	b	b	b
a	a	b	a	b	b	b	a
a	b	a	b	b	b	a	a
a	b	b	b	a	a	a	b
b	a	a	a	b	a	b	b
b	a	b	b	b	a	a	a
b	b	a	a	a	b	a	b
b	b	b	a	a	a	b	a

bwt(aaababbb) $=$ baabbaba

The BWT of a dB sequence

$t =$ aaababbb

							bwt(t)
a	a	a	b	a	b	b	b
a	a	b	a	b	b	b	a
a	b	a	b	b	b	a	a
a	b	b	b	a	a	a	b
b	a	a	a	b	a	b	b
b	a	b	b	b	a	a	a
b	b	a	a	a	b	a	b
b	b	b	a	a	a	b	a

The BWT of a dB sequence

$t =$ aaababbb

bwt(t)

a	a	a	b	a	b	b	b
a	a	b	a	b	b	b	a
a	b	a	b	b	b	a	a
a	b	b	b	a	a	a	b
b	a	a	a	b	a	b	b
b	a	b	b	b	a	a	a
b	b	a	a	a	b	a	b
b	b	b	a	a	a	b	a

$bwt(t) = u_0 u_1 \cdots u_{2^{k-1}-1}$, where each block $u_i \in \{ab, ba\}$

Question Is every string of the form $v \in \{\text{ab},\text{ba}\}^{2^{k-1}}$ the BWT of a dB sequence?

No! Ex. $v = \text{babababa}$, its standard perm. is

$$\pi_v = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 0 & 5 & 1 & 6 & 2 & 7 & 3 \end{smallmatrix}\right) = (0,4,6,7,3,1)(2,5)$$

The extended BWT (eBWT) is a generalization of the BWT, where every v is the eBWT of something (of a multiset of strings).

Ex. Here we get two strings, one for each cycle: $\{\text{aaabbb},\text{ab}\}$.

The extended BWT

Def. (Mantaci et al., 2007) Let $\mathcal{M}$ be a multiset of primitive strings. The extended BWT (eBWT) of $\mathcal{M}$ is the concatenation of the last characters of its rotations, taken in omega order.

$\mathcal{M} = \{$a,ab,aabbb$\}$

a	a
aabbb	b
ab	b
abbba	a
baabb	b
ba	a
bbaab	b
bbbaa	a

Def. (omega-order): $T <_\omega S$ if (i) $T^\omega <_{lex} S^\omega$, or (ii) $T^\omega = S^\omega$, $T = U^k, S = U^m$ and $k < m$.

The basic theorem

Thm (Higgins, 2012) $v \in \{ab, ba\}^{2^{k-1}}$ if and only if v is the eBWT of a de Bruijn set of order k.

Def. (Higgins, 2012) A binary de Bruijn set of order k is a multiset of total length 2^k such that every k-mer is the prefix of some rotation of some power of some string in $\mathcal{M}$.

Ex. $\mathcal{M}_1 = \{\text{aaabbb}, \text{ab}\}$, $\mathcal{M}_2 = \{\text{a}, \text{ab}, \text{aabbb}\}$.

Coro $v \in \{ab, ba\}^{2^{k-1}}$ is the BWT of a dB sequence if and only if π_v is cyclic.

Swapping characters in the eBWT

Lemma (Swap Lemma) Let $v \in \{\mathsf{a},\mathsf{b}\}^*, v_i \neq v_{i+1}$, and v' be the result of swapping v_i and v_{i+1}. If v_i and v_{i+1} belong to **distinct cycles** in the cycle decomposition of π_v then the number of cycles **decreases by one**; otherwise it **increases by one**.

Swapping characters in the eBWT

Lemma (Swap Lemma) Let $v \in \{\text{a,b}\}^*$, $v_i \neq v_{i+1}$, and v' be the result of swapping v_i and v_{i+1}. If v_i and v_{i+1} belong to **distinct cycles** in the cycle decomposition of π_v then the number of cycles **decreases by one**; otherwise it **increases by one**.

Ex.

$$v = \text{ba}\mathbf{ab}\text{baba} \qquad \pi_v = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 0 & \mathbf{1} & \mathbf{5} & 6 & 2 & 7 & 3 \end{smallmatrix}\right) = (0,4,6,7,3,5,2,1)$$

$$v' = \text{ba}\mathbf{ba}\text{baba} \qquad \pi_{v'} = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 0 & \mathbf{5} & \mathbf{1} & 6 & 2 & 7 & 3 \end{smallmatrix}\right) = (0,4,6,7,3,1)(2,5)$$

This is a generalization of a technique from [Giuliani, L., Masillo, Rizzi, 2021].

Transforming the eBWT of a dB set into the BWT of a dB sequence

v = abababab = $(ab)^4$

a	b	a	b	a	b	a	b
0	4	1	5	2	6	3	7
0	1	2	3	4	5	6	7
a	a	a	a	b	b	b	b

Transforming . . .

- $v = abababab = (ab)^4$

a	b	a	b	a	b	a	b
0	4	1	5	2	6	3	7
0	1	2	3	4	5	6	7
a	a	a	a	b	b	b	b

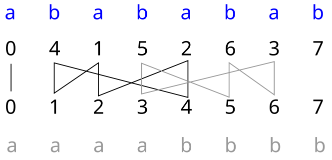
(0)

- v = abababab = $(ab)^4$

a	b	a	b	a	b	a	b
0	4	1	5	2	6	3	7
0	1	2	3	4	5	6	7
a	a	a	a	b	b	b	b

(0) (1 4 2)

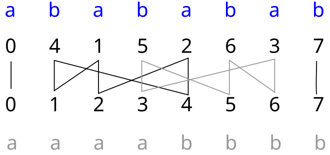
- $v = ababab ab = (ab)^4$



(0) (1 4 2) (3 5 6)

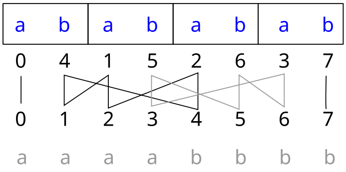
Transforming . . .

- v = abababab = $(ab)^4$



(0) (1 4 2) (3 5 6) (7)

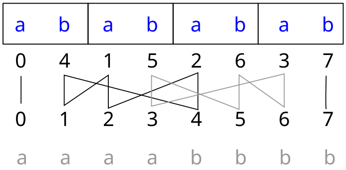
- $v = ababab ab = (ab)^4$



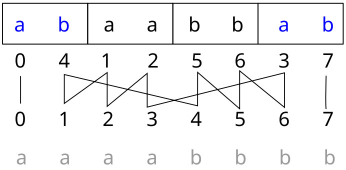
(0) (1 4 2) (3 5 6) (7)

Transforming . . .

- $v = abababab = (ab)^4$



(0) (1 4 2) (3 5 6) (7)

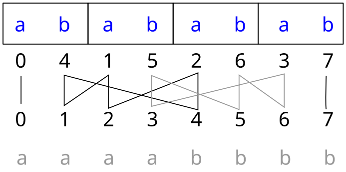


(0) (1 4 5 6 3 2) (7)

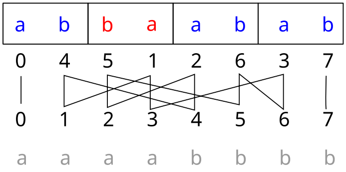
If we swap $(3, 4)$ then the resulting string is not in the set $\{ab, ba\}^{2^{k-1}}$. We show that it suffices to swap always within blocks.

Generation of binary de Bruijn sequences of order k

- v = abababab = $(ab)^4$



(0) (1 4 2) (3 5 6) (7)



(0) (1 4 2 5 6 3) (7)

We call a block unhappy if its elements are in different cycles. Here we have 4 unhappy blocks, but we need only 3 swaps to get one cycle.

a	b	b	a	a	b	a	b
0	4	5	1	2	6	3	7

0	1	2	3	4	5	6	7
a	a	a	a	b	b	b	b

(0) (1 4 2 5 6 3) (7)

b	a	b	a	a	b	a	b
4	0	5	1	2	6	3	7
0	1	2	3	4	5	6	7
a	a	a	a	b	b	b	b

(0 4 2 5 6 3 1) (7)

b	a	b	a	a	b	b	a
4	0	5	1	2	6	3	7
0	1	2	3	4	5	6	7
a	a	a	a	b	b	b	b

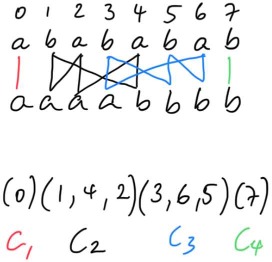
(0 4 2 5 6 7 3 1)

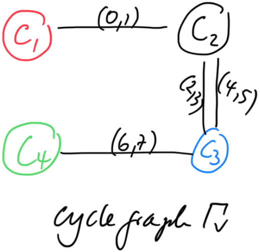
b	a	b	a	a	b	b	a
4	0	5	1	2	6	3	7
0	1	2	3	4	5	6	7
a	a	a	a	b	b	b	b

$$(0\quad 4\quad 2\quad 5\quad 6\quad 7\quad 3\quad 1)$$

- v = babaabba
- $\mathrm{bwt}^{-1}(v)$ = aaabbbab

How to choose the edges





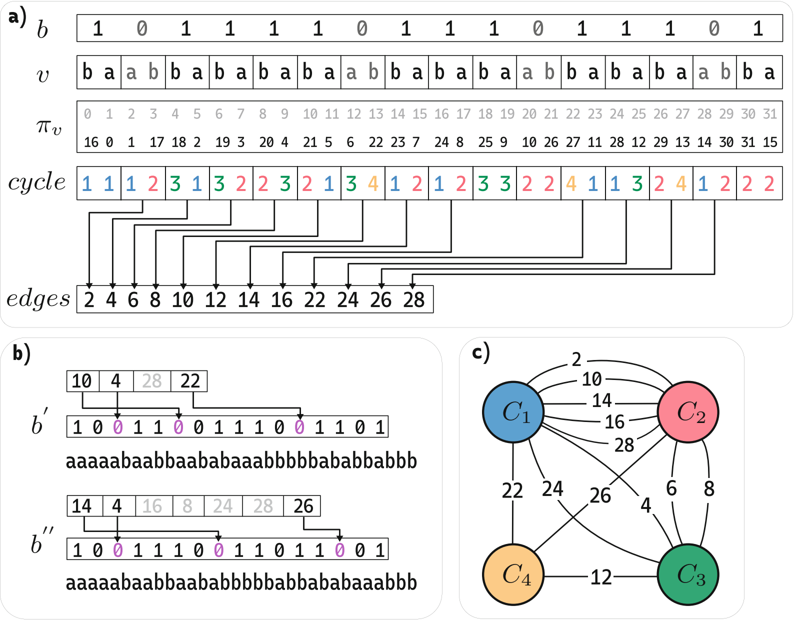
- cycle graph Γ_v: vertices = cycles, edges = unhappy blocks
- Spanning Trees (STs) of Γ_v = (BWTs of) dB sequences
- here: 2 STs = 2 dB seqs (aaabbbab, aaababbb)

Some final details

- The standard permutation can be computed easily: the ith block $\pi_v(\{2i, 2i+1\}) = \{i, n/2 + i\}$,
where $n = 2^k =$ length of dB seq. (no rank-function needed)
- We do not need v or t: replace ab $\mapsto 0$, ba $\mapsto 1$.
- enc(babaabba) $= 1101$, dec(1101) $=$ babaabba

Algorithm overview

1. Choose a random bitstring b of length 2^{k-1}.
2. Compute the standard permutation π_v of $v = dec(b)$.
3. Construct the cycle graph Γ_v.
4. Choose a random spanning tree T of Γ_v.
5. Flip the bits of b corresponding to T, resulting in b'.
6. Invert $s = dec(b')$, resulting in dB seq t.


a)
b
1 0 1 1 1 1 0 1 1 1 0 1 1 1 0 1
v
b a a b b a b a b a b a a b b a b a b a a b b a b a b a a b b a
π_v
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
16 0 1 17 18 2 19 3 20 4 21 5 6 22 23 7 24 8 25 9 10 26 27 11 28 12 29 13 14 30 31 15
cycle
1 1 1 2 3 1 3 2 2 3 2 1 3 4 1 2 1 2 3 3 2 2 4 1 1 3 2 4 1 2 2 2
edges
2 4 6 8 10 12 14 16 22 24 26 28
b)
10 4 28 22
b′
1 0 0 1 1 0 0 1 1 1 0 0 1 1 0 1
aaaaabaabbaababaaabbbbbababbabbb
14 4 16 8 24 28 26
b′′
1 0 0 1 1 1 0 0 1 1 0 1 1 0 0 1
aaaaabaabbaababbbbbabbababaaabbb
c)
C_1
C_2
C_3
C_4
2
10
14
16
28
22
24
26
4
6
8
12

Algorithm implementation and analysis

1. Choose a random bitstring b of length 2^{k-1}. $\mathcal{O}(n)$

Algorithm implementation and analysis

1. Choose a random bitstring b of length 2^{k-1}. $\mathcal{O}(n)$
2. Compute the standard permutation π_v of $v = dec(b)$.
   Fill in the cycle-array on the fly. $\mathcal{O}(n)$

Algorithm implementation and analysis

1. Choose a random bitstring b of length 2^{k-1}. $\mathcal{O}(n)$
2. Compute the standard permutation π_v of $v = dec(b)$.
   Fill in the cycle-array on the fly. $\mathcal{O}(n)$
3. Construct the cycle graph Γ_v.
   Compute the edges array. $\mathcal{O}(n)$

Algorithm implementation and analysis

1. Choose a random bitstring b of length 2^{k-1}. $\mathcal{O}(n)$
2. Compute the standard permutation π_v of $v = dec(b)$.
   Fill in the cycle-array on the fly. $\mathcal{O}(n)$
3. Construct the cycle graph Γ_v.
   Compute the edges array. $\mathcal{O}(n)$
4. Choose a random spanning tree T of Γ_v.
   Union-Find data structure, $|\Gamma_v|$ at most $Z_k = \sum_{d|k} Lyn(d)$
   $\alpha(n)$ inverse Ackerman function; $Z_k \sim 2^{k-1}/k = \Theta(n)$ $\mathcal{O}(n\alpha(n))$

Algorithm implementation and analysis

1. Choose a random bitstring b of length 2^{k-1}. $\mathcal{O}(n)$
2. Compute the standard permutation π_v of $v = dec(b)$.
   Fill in the cycle-array on the fly. $\mathcal{O}(n)$
3. Construct the cycle graph Γ_v.
   Compute the edges array. $\mathcal{O}(n)$
4. Choose a random spanning tree T of Γ_v.
   Union-Find data structure, $|\Gamma_v|$ at most $Z_k = \sum_{d|k} Lyn(d)$
   $\alpha(n)$ inverse Ackerman function; $Z_k \sim 2^{k-1}/k = \Theta(n)$ $\mathcal{O}(n\alpha(n))$
5. Flip the bits of b corresponding to T, resulting in b'. $\mathcal{O}(n)$
   (We actually do 5 in parallel with 4.)

Algorithm implementation and analysis

1. Choose a random bitstring b of length 2^{k-1}. $\mathcal{O}(n)$
2. Compute the standard permutation π_v of $v = dec(b)$.
   Fill in the cycle-array on the fly. $\mathcal{O}(n)$
3. Construct the cycle graph Γ_v.
   Compute the edges array. $\mathcal{O}(n)$
4. Choose a random spanning tree T of Γ_v.
   Union-Find data structure, $|\Gamma_v|$ at most $Z_k = \sum_{d|k} Lyn(d)$
   $\alpha(n)$ inverse Ackerman function; $Z_k \sim 2^{k-1}/k = \Theta(n)$ $\mathcal{O}(n\alpha(n))$
5. Flip the bits of b corresponding to T, resulting in b'. $\mathcal{O}(n)$
   (We actually do 5 in parallel with 4.)
6. Invert $s = dec(b')$, resulting in dB sequence t. $\mathcal{O}(n)$

Algorithm implementation and analysis

1. Choose a random bitstring b of length 2^{k-1}. $\mathcal{O}(n)$
2. Compute the standard permutation π_v of $v = dec(b)$.
   Fill in the cycle-array on the fly. $\mathcal{O}(n)$
3. Construct the cycle graph Γ_v.
   Compute the edges array. $\mathcal{O}(n)$
4. Choose a random spanning tree T of Γ_v.
   Union-Find data structure, $|\Gamma_v|$ at most $Z_k = \sum_{d|k} Lyn(d)$
   $\alpha(n)$ inverse Ackerman function; $Z_k \sim 2^{k-1}/k = \Theta(n)$ $\mathcal{O}(n\alpha(n))$
5. Flip the bits of b corresponding to T, resulting in b'. $\mathcal{O}(n)$
   (We actually do 5 in parallel with 4.)
6. Invert $s = dec(b')$, resulting in dB sequence t. $\mathcal{O}(n)$

total running time $\mathcal{O}(n\alpha(n))$
space $\mathcal{O}(n)$

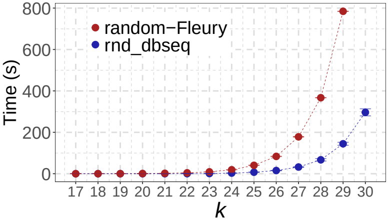
Running time

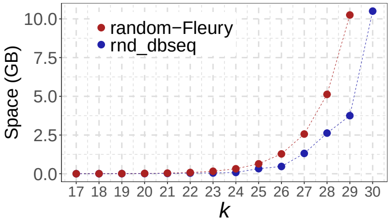
k	17	18	19	20	21	22	23	24	25	26	27	28	29	30
w/o (s)	0.003	0.01	0.02	0.04	0.10	0.29	0.87	2.63	6.07	12.42	27.49	57.19	125.38	247.10
w (s)	0.01	0.02	0.03	0.07	0.16	0.39	0.96	3.11	7.31	15.44	32.32	67.20	144.72	293.49

Average running time in seconds, taken over 100 randomly generated dB sequences, without (w/o) and with (w) the time for outputting the dB sequence, on a laptop with 16 GB of RAM.

Comparison with Fleury's algorithm

- We modified an implementation of Fleury's algorithm from `debruijnsequence.org` → random-Fleury
- random-Fleury cannot construct all possible dB seqs, but serves as the closest available method for comparison





Our algorithm is appr. 10-12 times faster for $17 \leq k \leq 23$, and 5 times faster for $k = 29$, and uses only half the memory.

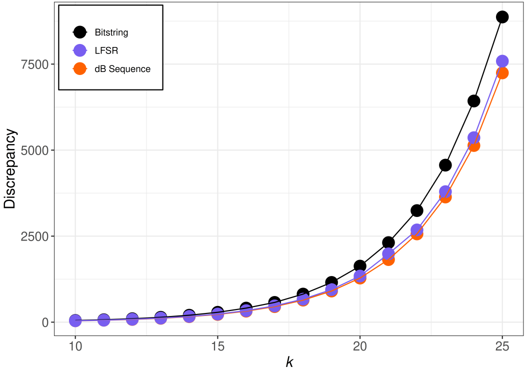
Def. The discrepancy of a binary string is the maximum absolute difference between the number of a's and b's over all (circular) substrings.

- **Low discrepancy is preferable** for certain applications

AAAAAABAAAABBAAABABAAABBBAABAABABBAABBABAABBBBABABABBBABBABBBBBB

|#A - #B| = 17 - 5 = 12

Estimating the average discrepancy of dB sequences



Average discrepancy of LFSRs from (Gabric and Sawada, 2022).

- For studying properties of de Bruijn sequences, not realistic to use random bitstrings or LFSRs as a sample.

Not uniformly at random

Our algorithm does not output all dB sequences according to the uniform probability distribution, for two reasons:

1. the ST of the cycle graph is not chosen uniformly at random
2. even if it was, not every dB sequence would be equally likely to be output

Not uniformly at random

Our algorithm does not output all dB sequences according to the uniform probability distribution, for two reasons:

1. the ST of the cycle graph is not chosen uniformly at random
2. even if it was, not every dB sequence would be equally likely to be output

ad 1 Fastest algorithms for choosing a ST of a multigraph uniformly at random run in superquadratic time (Dufree et al., STOC 2017)

Not uniformly at random

Our algorithm does not output all dB sequences according to the uniform probability distribution, for two reasons:

1. the ST of the cycle graph is not chosen uniformly at random
2. even if it was, not every dB sequence would be equally likely to be output

ad 1 Fastest algorithms for choosing a ST of a multigraph uniformly at random run in superquadratic time (Dufree et al., STOC 2017)

ad 2 We define the prestige of a dB sequence t as

$$pres(t) = \frac{1}{2^{2^{k-1}}} \sum_{v \in \{\mathrm{ab},\mathrm{ba}\}^{2^{k-1}}} p(t \mid v)$$

Not uniformly at random

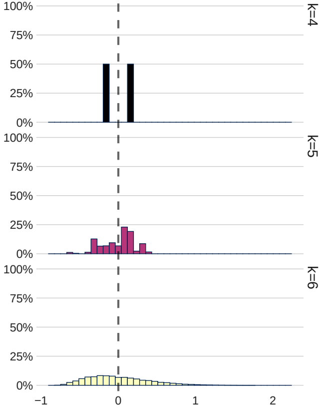


Figure: Comparison of empirical probabilities (left) and prestige (right) to the uniform distribution (vertical line), for $k = 4, 5, 6$. y-axis: % of dB seqs that share the same P_e resp. prestige. x-axes normalized w.r.t. P_u.

Conclusion

- first practical algorithm for constructing a random dB sequence which produces *any* dB sequence with positive probability
  - time $\mathcal{O}(n\alpha(n))$
  - space $\mathcal{O}(n)$

Conclusion

- first practical algorithm for constructing a random dB sequence which produces *any* dB sequence with positive probability
  - time $\mathcal{O}(n\alpha(n))$
  - space $\mathcal{O}(n)$
- implementation: `github.com/lucaparmigiani/rnd_dbseq`
  - simple (less than 120 lines of C++ code)
  - fast (less than one second on a laptop for k up to 23)

Conclusion

- first practical algorithm for constructing a random dB sequence which produces *any* dB sequence with positive probability
  - time $\mathcal{O}(n\alpha(n))$
  - space $\mathcal{O}(n)$
- implementation: `github.com/lucaparmigiani/rnd_dbseq`
  - simple (less than 120 lines of C++ code)
  - fast (less than one second on a laptop for k up to 23)
- try it: `debruijnsequence.org/db/random`

Conclusion

- first practical algorithm for constructing a random dB sequence which produces *any* dB sequence with positive probability
  - time $\mathcal{O}(n\alpha(n))$
  - space $\mathcal{O}(n)$
- implementation: `github.com/lucaparmigiani/rnd_dbseq`
  - simple (less than 120 lines of C++ code)
  - fast (less than one second on a laptop for k up to 23)
- try it: `debruijnsequence.org/db/random`
- we improved the estimates for the average discrepancy of binary dB sequences

Conclusion

- first practical algorithm for constructing a random dB sequence which produces *any* dB sequence with positive probability
  - time $\mathcal{O}(n\alpha(n))$
  - space $\mathcal{O}(n)$
- implementation: `github.com/lucaparmigiani/rnd_dbseq`
  - simple (less than 120 lines of C++ code)
  - fast (less than one second on a laptop for k up to 23)
- try it: `debruijnsequence.org/db/random`
- we improved the estimates for the average discrepancy of binary dB sequences
- our algorithm can be straighforwardly extended to any constant-size alphabet (present on github)

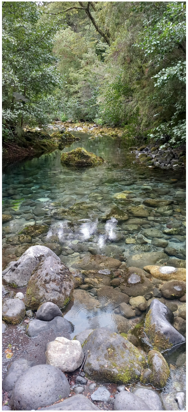
Open problems

- distribution of prestige (for rejection sampling)
- for $\sigma > 2$ a straightforward extension of our algorithm has running time $\mathcal{O}(\sigma n \alpha(n))$, due to up to $\binom{\sigma}{2}$ edges in each block; can this be improved?
- algorithm for uniformly random dB sequences

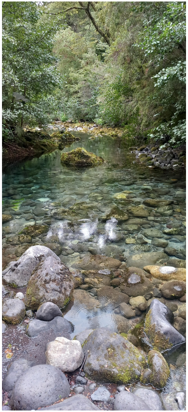
- paper:
  Proc. of LATIN2024
  (Puerto Varas, Chile,
  18-22 March 2024)

- code at (C++ and python):
  github.com/lucaparmigiani/rnd_dbseq

- try it at:
  debruijnsequences.org
  (website by Joe Sawada)



- paper:
  Proc. of LATIN2024
  (Puerto Varas, Chile,
  18-22 March 2024)
- code at (C++ and python):
  github.com/lucaparmigiani/rnd_dbseq
- try it at:
  debruijnsequences.org
  (website by Joe Sawada)



Thank you for your attention!