How to use the BWT to construct random de Bruijn sequences

Zsuzsanna Lipták

University of Verona (Italy)

SeqBIM 2024
Rennes, Nov. 29, 2024

The Burrows-Wheeler Transform (BWT)

Recall: $T =$ `banana`. The BWT is a permutation of T: `nnbaaa`

all rotations (conjugates)		all rotations, sorted
		L
banana	$\xrightarrow[\text{lexicographic order}]{}$	abanan
ananab		anaban
nanaba		ananab
anaban		banana
nabana		nabana
abanan		nanaba

The Burrows-Wheeler Transform (BWT)

Recall: $T =$ `banana`. The BWT is a permutation of T: `nnbaaa`

all rotations (conjugates)

`banana`
`ananab`
`nanaba`
`anaban`
`nabana`
`abanan`

$\longrightarrow$
lexicographic
order

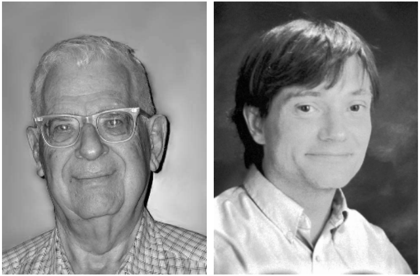
all rotations, sorted

L
`abanan`
`anaban`
`ananab`
`banana`
`nabana`
`nanaba`

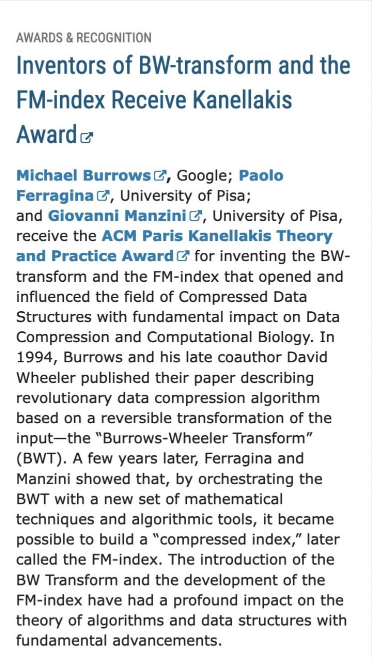
BWT(T) = concatenation of last characters = L

The Burrows-Wheeler Transform

- introduced by Burrows and Wheeler in 1994
- a reversible string transform
- basis of a highly effective lossless text compression algorithm
- basis of compressed data structures (compressed text indexes)



source: Adjeroh, Bell, Mukerjee (2008)

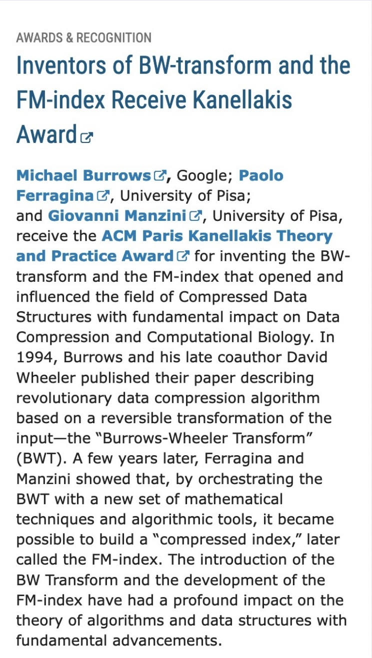


AWARDS & RECOGNITION

Inventors of BW-transform and the FM-index Receive Kanellakis Award

Michael Burrows, Google; **Paolo Ferragina**, University of Pisa; and **Giovanni Manzini**, University of Pisa, receive the **ACM Paris Kanellakis Theory and Practice Award** for inventing the BW-transform and the FM-index that opened and influenced the field of Compressed Data Structures with fundamental impact on Data Compression and Computational Biology. In 1994, Burrows and his late coauthor David Wheeler published their paper describing revolutionary data compression algorithm based on a reversible transformation of the input—the "Burrows-Wheeler Transform" (BWT). A few years later, Ferragina and Manzini showed that, by orchestrating the BWT with a new set of mathematical techniques and algorithmic tools, it became possible to build a "compressed index," later called the FM-index. The introduction of the BW Transform and the development of the FM-index have had a profound impact on the theory of algorithms and data structures with fundamental advancements.

- 2022 ACM Kanellakis Theory and Practice Award
- for BWT and FM-index (Ferragina & Manzini 2000, 2005)

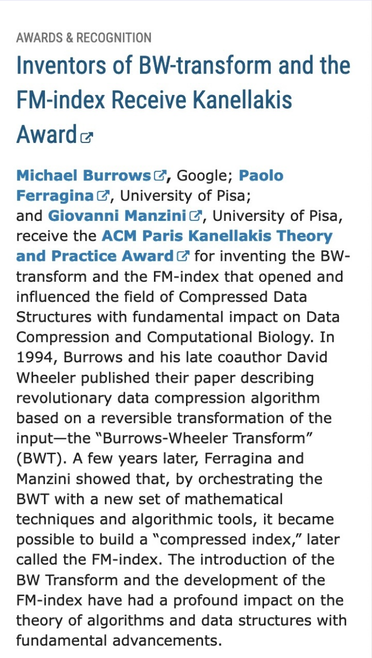


AWARDS & RECOGNITION

Inventors of BW-transform and the FM-index Receive Kanellakis Award

Michael Burrows, Google; **Paolo Ferragina**, University of Pisa; and **Giovanni Manzini**, University of Pisa, receive the **ACM Paris Kanellakis Theory and Practice Award** for inventing the BW-transform and the FM-index that opened and influenced the field of Compressed Data Structures with fundamental impact on Data Compression and Computational Biology. In 1994, Burrows and his late coauthor David Wheeler published their paper describing revolutionary data compression algorithm based on a reversible transformation of the input—the "Burrows-Wheeler Transform" (BWT). A few years later, Ferragina and Manzini showed that, by orchestrating the BWT with a new set of mathematical techniques and algorithmic tools, it became possible to build a "compressed index," later called the FM-index. The introduction of the BW Transform and the development of the FM-index have had a profound impact on the theory of algorithms and data structures with fundamental advancements.

- 2022 ACM Kanellakis Theory and Practice Award
- for BWT and FM-index (Ferragina & Manzini 2000, 2005)
- *"... that opened and influenced the field of Compressed Data Structures with fundamental impact on Data Compression and Computational Biology"*



AWARDS & RECOGNITION

Inventors of BW-transform and the FM-index Receive Kanellakis Award

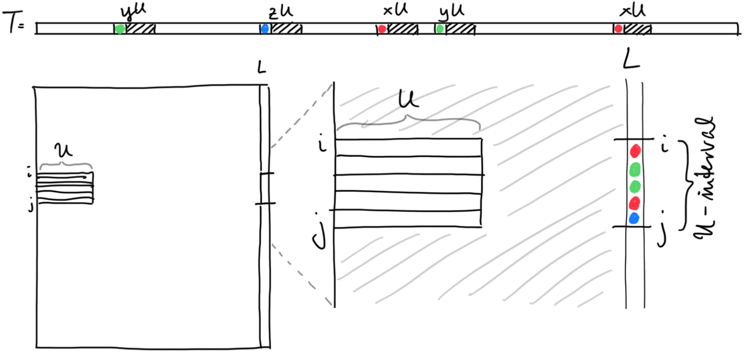
Michael Burrows, Google; **Paolo Ferragina**, University of Pisa; and **Giovanni Manzini**, University of Pisa, receive the **ACM Paris Kanellakis Theory and Practice Award** for inventing the BW-transform and the FM-index that opened and influenced the field of Compressed Data Structures with fundamental impact on Data Compression and Computational Biology. In 1994, Burrows and his late coauthor David Wheeler published their paper describing revolutionary data compression algorithm based on a reversible transformation of the input—the "Burrows-Wheeler Transform" (BWT). A few years later, Ferragina and Manzini showed that, by orchestrating the BWT with a new set of mathematical techniques and algorithmic tools, it became possible to build a "compressed index," later called the FM-index. The introduction of the BW Transform and the development of the FM-index have had a profound impact on the theory of algorithms and data structures with fundamental advancements.

- 2022 ACM Kanellakis Theory and Practice Award
- for BWT and FM-index
  (Ferragina & Manzini 2000, 2005)
- *"... that opened and influenced the field of Compressed Data Structures with fundamental impact on Data Compression and Computational Biology"*
- some bioinformatics tools:
  - `bwa, bwa-sw, bwa-mem`
    (Li & Durbin, 2009, 2010, Li 2013)
    $>$ 55,000 cit.
  - `bowtie, bowtie2`
    (Langmead et al., 2009, 2012)
    $>$ 70,000 cit.

Some BWT technicalities

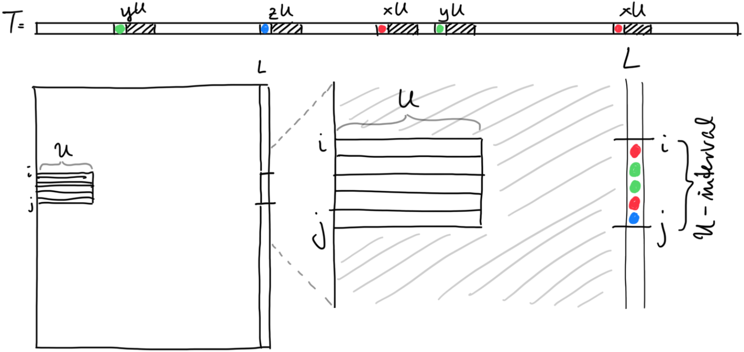
1. *U*-intervals

Def. Let U be a substring of T. We call $[i,j]$ the U-interval of $L = \text{bwt}(T)$, where the conjugates in positions $k \in [i,j]$ are exactly those starting with U:



1. *U*-intervals

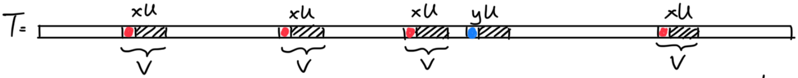
Def. Let U be a substring of T. We call $[i,j]$ the U-interval of $L = \text{bwt}(T)$, where the conjugates in positions $k \in [i,j]$ are exactly those starting with U:



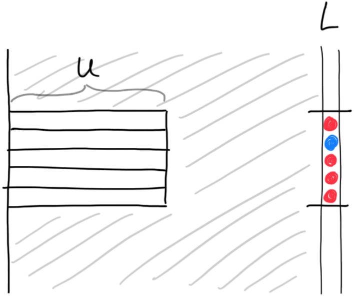
CA = conjugate array

Terminology: $L[i..j]$ = left-context of U; $[i,j] \cong$ SA-interval of U (here: CA)

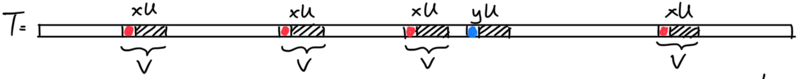
Why is the BWT so good in compression?



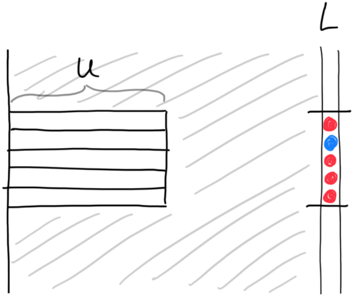
many occurrences of $V = xU$ $\Rightarrow$ many x's in U-interval



Why is the BWT so good in compression?

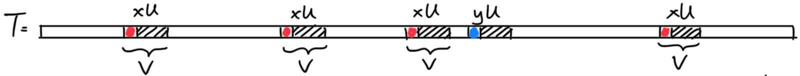


many occurrences of $V = xU$ ⇒ many x's in U-interval

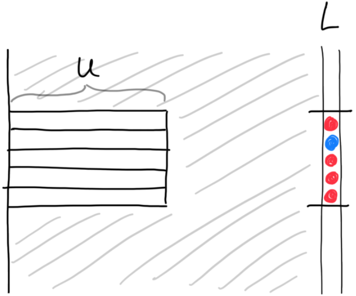


- T has many repeated substrings ⇒ many U-intervals mostly same character
- $L = \text{bwt}(T)$ has few runs ⇒ runlength encoding (RLE) is good

Why is the BWT so good in compression?



many occurrences of $V = xU \Rightarrow$ many x's in U-interval



- T has many repeated substrings $\Rightarrow$ many U-intervals mostly same character
- $L = \text{bwt}(T)$ has few runs $\Rightarrow$ runlength encoding (RLE) is good

$\texttt{bbbaccccccccccccccccccaaaaa} \mapsto \texttt{b}^3\texttt{a}^1\texttt{c}^{18}\texttt{a}^5$

An example: the U-interval for $U = \mathtt{he}+\mathtt{emptyspace}$ in an English text

rotation	BWT
...	
he caverns measureless to man, And sank in tumult to a ...	t
he caves. It was a miracle of rare device, A sunny pleasure-...	t
he dome of pleasure Floated midway on the waves; Where was ...	t
he fountain and the caves. It was a miracle of rare device,...	t
he green hill athwart a cedarn cover! A savage place! as ...	t
he hills, Enfolding sunny spots of greenery. But oh! that ...	t
he milk of Paradise.	t
he mingled measure From the fountain and the caves. It was a ...	t
he on honey-dew hath fed, And drunk the milk of Paradise. ...	␣
he played, Singing of Mount Abora. Could I revive within me ...	s
he sacred river ran, Then reached the caverns measureless ...	t
he sacred river, ran Through caverns measureless to man ...	t
he sacred river. Five miles meandering with a mazy motion ...	t
he shadow of the dome of pleasure Floated midway on the waves ...	T
he thresher's flail: And mid these dancing rocks at once and ...	t
he waves; Where was heard the mingled measure From the ...	t

Kubla Kahn by Samuel Coleridge
(1998 characters)

An example: the U-interval for $U =$ `he+emptyspace` in an English text

rotation	BWT
...	
`he caverns measureless to man, And sank in tumult to a ...`	t
`he caves. It was a miracle of rare device, A sunny pleasure-...`	t
`he dome of pleasure Floated midway on the waves; Where was ...`	t
`he fountain and the caves. It was a miracle of rare device,...`	t
`he green hill athwart a cedarn cover! A savage place! as ...`	t
`he hills, Enfolding sunny spots of greenery. But oh! that ...`	t
`he milk of Paradise.`	t
`he mingled measure From the fountain and the caves. It was a ...`	t
`he on honey-dew hath fed, And drunk the milk of Paradise. ...`	␣
`he played, Singing of Mount Abora. Could I revive within me ...`	s
`he sacred river ran, Then reached the caverns measureless ...`	t
`he sacred river, ran Through caverns measureless to man ...`	t
`he sacred river. Five miles meandering with a mazy motion ...`	t
`he shadow of the dome of pleasure Floated midway on the waves ...`	T
`he thresher's flail: And mid these dancing rocks at once and ...`	t
`he waves; Where was heard the mingled measure From the ...`	t

many `the`'s, some `he,` `she,` `The`

Kubla Kahn by Samuel Coleridge
(1998 characters)

2. The extended BWT

(Mantaci, Restivo, Rosone, Sciortino, TCS, 2007)

Ex. $\mathcal{M} = \{\texttt{bana}, \texttt{na}\}$. The eBWT is a permutation of the characters of $\mathcal{M}$: eBWT($\mathcal{M}$) $= \texttt{nbnaaa}$.

all rotations (conjugates)

bana
anab
naba
aban
na
an

$\longrightarrow$

omega order

all rotations, sorted

aban	n
anab	b
an	n
bana	a
naba	a
na	a

N.B. $\texttt{anab} <_\omega \texttt{an}$, since $\texttt{anab} \cdot \texttt{anab} \cdots <_{\text{lex}} \texttt{an} \cdot \texttt{an} \cdot \texttt{an} \cdot \texttt{an} \cdots$

The extended BWT (cont.)

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

$\mathcal{M} = \{\texttt{bana}, \texttt{na}\}$

omega-order		lex-order	
`aban`	`n`	`aban`	`n`
`anab`	`b`	`an`	`n`
`an`	`n`	`anab`	`b`
`bana`	`a`	`bana`	`a`
`naba`	`a`	`na`	`a`
`na`	`a`	`naba`	`a`

(**N.B.** With the lex-order, the LF-property would not hold!)

The extended BWT (cont.)

- omega-order instead of lex-order
- the eBWT inherits BWT properties: clustering effect, reversibility, useful for lossless text compression, efficient pattern matching, . . .
- However, until recently no linear-time algorithm was known.

The extended BWT (cont.)

- omega-order instead of lex-order
- the eBWT inherits BWT properties: clustering effect, reversibility, useful for lossless text compression, efficient pattern matching, ...
- However, until recently no linear-time algorithm was known.

Since 2021: linear-time algorithms and implementations available

- First linear-time algorithm (Bannai, Kärkkäinen, Köppl, Piatkowski, CPM 2021)
- We significantly simplified this algorithm (Boucher, Cenzato, L., Rossi, Sciortino, SPIRE 2021)
- ...and gave efficient implementations of the eBWT (`cais`,`pfpebwt` 2021)
- Later we gave an *r*-index based on the eBWT (—, Inf. & Comp. 2024)
- Recently, another linear-time algorithm for the eBWT has appeared (Olbrich, Ohlebusch, Büchler, ACM Tr Alg 2024)

3. The standard permutation

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 $i < j$.

In other words, π_V is a stable sort of the characters of V.

Example: $V =$ `nnbaaa`

0	1	2	3	4	5
n	n	b	a	a	a
a	a	a	b	n	n
0	1	2	3	4	5

$$\pi_V = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 3 & 0 & 1 & 2 \end{pmatrix}$$
$$= (0, 4, 1, 5, 2, 3)$$

(If V is a BWT, then π_V is called LF-mapping.)

The standard permutation (cont.)

- If V is a BWT, then π_V is called LF-mapping.

The standard permutation (cont.)

- If V is a BWT, then π_V is called LF-mapping.
- With π_V we can recover (a conjugate of) T from bwt(T) back-to-front:

  Ex. $V = \texttt{nnbaaa}$, $\pi_V = (0, 4, 1, 5, 2, 3)$

The standard permutation (cont.)

- If V is a BWT, then π_V is called LF-mapping.
- With π_V we can recover (a conjugate of) T from bwt(T) back-to-front:

Ex. $V = \texttt{nnbaaa}$, $\pi_V = (0, 4, 1, 5, 2, 3)$ `abanan`

The standard permutation (cont.)

- If V is a BWT, then π_V is called LF-mapping.
- With π_V we can recover (a conjugate of) T from bwt(T) back-to-front:

  Ex. $V =$ `nnbaaa`, $\pi_V = (0, 4, 1, 5, 2, 3)$ `abanan`
  (or given pos. 3: `banana`)

The standard permutation (cont.)

- If V is a BWT, then π_V is called LF-mapping.
- With π_V we can recover (a conjugate of) T from bwt(T) back-to-front:

  Ex. $V =$ `nnbaaa`, $\pi_V = (0, 4, 1, 5, 2, 3)$ `abanan`
  (or given pos. 3: `banana`)
- Similarly, we can recover (conjugates of) $\mathcal{M}$ from eBWT($\mathcal{M}$):

  Ex. $V =$ `nbnaaa`, $\pi_V = (0, 4, 1, 3)(2, 5)$

The standard permutation (cont.)

- If V is a BWT, then π_V is called LF-mapping.
- With π_V we can recover (a conjugate of) T from bwt(T) back-to-front:

  Ex. $V =$ `nnbaaa`, $\pi_V = (0, 4, 1, 5, 2, 3)$ `abanan`
  (or given pos. 3: `banana`)
- Similarly, we can recover (conjugates of) $\mathcal{M}$ from eBWT($\mathcal{M}$):

  Ex. $V =$ `nbnaaa`, $\pi_V = (0, 4, 1, 3)(2, 5)$ `aban, an`

The standard permutation (cont.)

- If V is a BWT, then π_V is called LF-mapping.
- With π_V we can recover (a conjugate of) T from $\text{bwt}(T)$ back-to-front:

  Ex. $V =$ `nnbaaa`, $\pi_V = (0, 4, 1, 5, 2, 3)$ `abanan`
  (or given pos. 3: `banana`)
- Similarly, we can recover (conjugates of) $\mathcal{M}$ from $\text{eBWT}(\mathcal{M})$:

  Ex. $V =$ `nbnaaa`, $\pi_V = (0, 4, 1, 3)(2, 5)$ `aban, an`
  (or given the positions $3, 4$: `bana, na`)

The standard permutation (cont.)

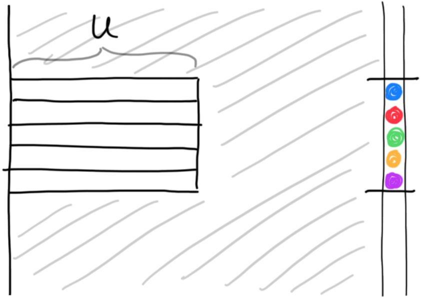
- If V is a BWT, then π_V is called LF-mapping.
- With π_V we can recover (a conjugate of) T from $\mathrm{bwt}(T)$ back-to-front:

  Ex. $V =$ `nnbaaa`, $\pi_V = (0, 4, 1, 5, 2, 3)$ `abanan`
  (or given pos. 3: `banana`)
- Similarly, we can recover (conjugates of) $\mathcal{M}$ from $\mathrm{eBWT}(\mathcal{M})$:

  Ex. $V =$ `nbnaaa`, $\pi_V = (0, 4, 1, 3)(2, 5)$ `aban, an`
  (or given the positions $3, 4$: `bana, na`)

Thm. (Folklore) A string V is the BWT of a primitive string if and only if π_V is cyclic.

Generating random de Bruijn sequences



joint work with Luca Parmigiani

de Bruijn sequences

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

k-mer = string of length k

Ex. $k = 3$: $\underset{0\,1\,2\,3\,4\,5\,6\,7}{\texttt{aaababbb}}$ (binary)

de Bruijn sequences

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

k-mer $=$ string of length k

Ex. $k = 3$: $\underset{0\,1\,2\,3\,4\,5\,6\,7}{\texttt{aaababbb}}$ (binary)

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

de Bruijn sequences

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

k-mer = string of length k

Ex. $k = 3$: $\underset{0\,1\,2\,3\,4\,5\,6\,7}{\texttt{aaababbb}}$ (binary)

$k = 3$: `aaacaabbabcacccabacbccbbbcb` (ternary)

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

de Bruijn sequences

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

k-mer = string of length k

Ex. $k = 3$: $\underset{0\,1\,2\,3\,4\,5\,6\,7}{\texttt{aaababbb}}$ (binary)

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

$k = 3$: `aaacaabbabcacccabacbccbbbcb` (ternary)

Easy: length of a dB sequence is σ^k ($\sigma = |\Sigma|$)

de Bruijn sequences

- de Bruijn sequences exist for every k and σ

de Bruijn sequences

- de Bruijn sequences exist for every k and σ
- There are $(\sigma!)^{\sigma^{k-1}}/\sigma^k$ dB sequences of order k

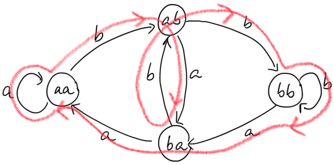
(Fly Sainte-Marie 1894, Tatyana van Aardenne-Ehrenfest and Nicolaas de Bruijn 1951: BEST Thm.)

de Bruijn sequences

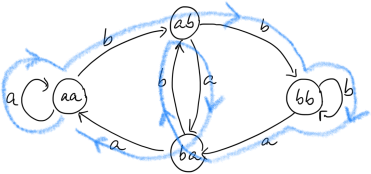
- de Bruijn sequences exist for every k and σ
- There are $(\sigma!)^{\sigma^{k-1}}/\sigma^k$ dB sequences of order k

(Fly Sainte-Marie 1894,
Tatyana van Aardenne-Ehrenfest and Nicolaas de Bruijn 1951: BEST Thm.)

- dB sequences correspond to Euler cycles in the dB graph.
  Ex.: $\sigma = 2, k = 3$:

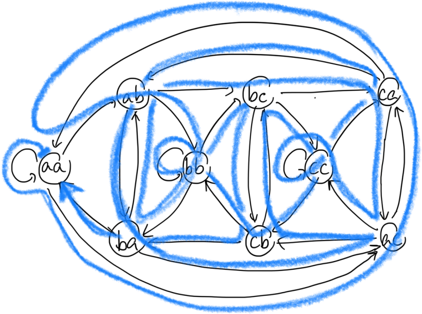


aaababbb

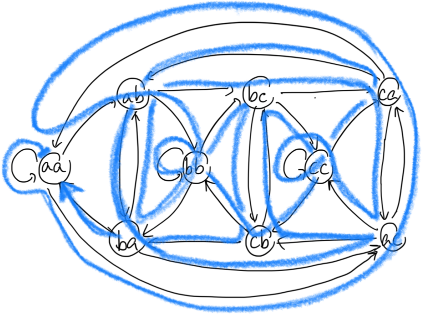


aaabbbab

Example for $\sigma = 3, k = 3$: `aaacaabbabcacccabacbccbbbcb`



Example for $\sigma = 3, k = 3$: `aaacaabbabcacccabacbccbbbcb`



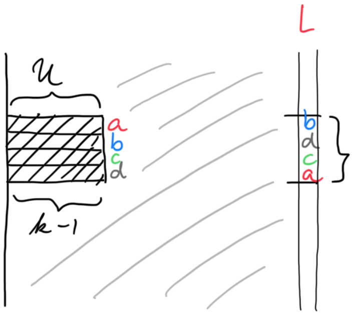
N.B. This is one of the 373 248 dB seqs for $\sigma = 3, k = 3$.

(number of dB seqs = $(\sigma!)^{\sigma^{k-1}}/\sigma^k$)

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
- cryptographic protocols
- ...

The BWT of de Bruijn sequences

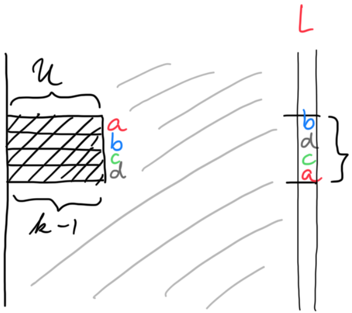


$u \in \Sigma^{k-1}$ occurs σ times

u-interval contains a permutation of Σ

In particular, BWT+RLE does not compress well: many runs!

The BWT of de Bruijn sequences



$U \in \Sigma^{k-1}$ occurs σ times

U-interval contains a permutation of Σ

In particular, BWT+RLE does not compress well: many runs!

N.B. From now on: binary dB sequences (for simplicity).

Construction algorithms

Many algorithms for constructing dB sequences. Some good overviews:

- H. Fredricksen: *A survey of full length nonlinear shift register cycle algorithms*, 1982 (classic survey)
- Chang et al., SN Computer Sc., 2021
- Gabric & Sawada, Discr. Math. 2022
- website `debruijnsequence.org` run by Joe Sawada and others

Construction algorithms

Many algorithms for constructing dB sequences. Some good overviews:

- H. Fredricksen: *A survey of full length nonlinear shift register cycle algorithms*, 1982 (classic survey)
- Chang et al., SN Computer Sc., 2021
- Gabric & Sawada, Discr. Math. 2022
- website `debruijnsequence.org` run by Joe Sawada and others

Most construct:

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

Construction algorithms

Many algorithms for constructing dB sequences. Some good overviews:

- H. Fredricksen: *A survey of full length nonlinear shift register cycle algorithms*, 1982 (classic survey)
- Chang et al., SN Computer Sc., 2021
- Gabric & Sawada, Discr. Math. 2022
- website `debruijnsequence.org` run by Joe Sawada and others

Most construct:

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

Construction algorithms (cont.)

- Linear feedback shift registers (LFSRs):

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

- number of binary dB sequences $= 2^{2^{k-1}-k}$
- 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, no practical algorithms exist for random dB sequence construction that can output any dB sequence with positive probability.

Construction of random dB sequences

- Surprisingly, no practical algorithms exist for random dB sequence construction that can output any dB sequence with positive probability.
- Our algorithm does just that!

Construction of random dB sequences

- Surprisingly, no practical algorithms exist for random dB sequence construction that can output any dB sequence with positive probability.
- Our algorithm does just that!
- . . . in near-linear time $\mathcal{O}(n\alpha(n))$, $n =$ length of dB sequence
  $\alpha =$ inverse Ackermann function

Construction of random dB sequences

- Surprisingly, no practical algorithms exist for random dB sequence construction that can output any dB sequence with positive probability.
- Our algorithm does just that!
- . . . in near-linear time $\mathcal{O}(n\alpha(n))$, $n =$ length of dB sequence
  $\alpha =$ inverse Ackermann function
- . . . and it is beautifully simple at that!

The BWT of a dB sequence

$T = \texttt{aaababbb}, k = 3$

$$\begin{array}{cccccccc}
a & a & a & b & a & b & b & b \\
a & a & b & a & b & b & b & a \\
a & b & a & b & b & b & a & a \\
a & b & b & b & a & a & a & b \\
b & a & a & a & b & a & b & b \\
b & a & b & b & b & a & a & a \\
b & b & a & a & a & b & a & b \\
b & b & b & a & a & a & b & a
\end{array}$$

$\text{bwt}(\texttt{aaababbb}) = \texttt{baabbaba}$

The BWT of a dB sequence

$T = \texttt{aaababbb}, k = 3$

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

$\mathrm{bwt}(\texttt{aaababbb}) = \texttt{baabbaba}$

Obs: $\mathrm{bwt}(T) \in \{\texttt{ab}, \texttt{ba}\}^{2^{k-1}}$

The BWT of a dB sequence

$T = \texttt{aaababbb}, k = 3$

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

$\text{bwt}(\texttt{aaababbb}) = \texttt{baabbaba}$

Obs: $\text{bwt}(T) \in \{\texttt{ab}, \texttt{ba}\}^{2^{k-1}}$

Proof: Every $(k-1)$-mer occurs exactly twice, preceded once by a, once by b.

The BWT of a dB sequence (cont.)

Q. Is every string $V \in \{\texttt{ab},\texttt{ba}\}^{2^{k-1}}$ the BWT of a dB sequence?

The BWT of a dB sequence (cont.)

Q. Is every string $V \in \{\mathtt{ab},\mathtt{ba}\}^{2^{k-1}}$ the BWT of a dB sequence?

A. No! e.g. $V = \mathtt{abbababa}$, its standard permutation is

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

Indeed, $V = \text{eBWT}(\{\mathtt{a}, \mathtt{aabbb}, \mathtt{ab}\})$.

The BWT of a dB sequence (cont.)

Q. Is every string $V \in \{\mathtt{ab}, \mathtt{ba}\}^{2^{k-1}}$ the BWT of a dB sequence?

A. No! e.g. $V = \mathtt{abbababa}$, its standard permutation is

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

Indeed, $V = \text{eBWT}(\{\mathtt{a}, \mathtt{aabbb}, \mathtt{ab}\})$.

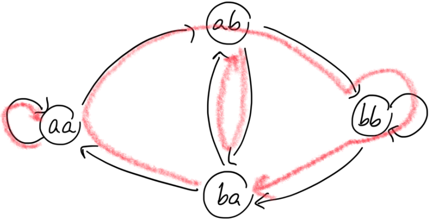
Def. (Higgins, 2012) A binary de Bruijn set of order k is a multiset of total length 2^k such that every k-mer is the prefix of some rotation of some power of some string in $\mathcal{M}$.

Ex. $\mathcal{M} = \{\mathtt{a}, \mathtt{aabbb}, \mathtt{ab}\}$ k-mers: $\mathtt{aaa}, \mathtt{aab}, \mathtt{bab}, \ldots$

dB sets

Indeed, dB sets correspond to edge cycle covers of the dB graph

Ex.: $\mathcal{M} = \{\texttt{a}, \texttt{aabbb}, \texttt{ab}\}$



The basic theorem

Thm (Higgins, 2012) The set $\{\texttt{ab},\texttt{ba}\}^{2^{k-1}}$ is the set of eBWTs of binary de Bruijn sets of order k.

Corollary A string $V \in \{\texttt{ab},\texttt{ba}\}^{2^{k-1}}$ is the BWT of a dB sequence if and only if π_V is cyclic.

Our idea: Take a random $V \in \{\texttt{ab}, \texttt{ba}\}^{2^{k-1}}$ and turn it into the BWT of a dB sequence.

Lemma (Swap Lemma) Let V be a binary string, $V_i \neq V_{i+1}$, and V' the result of swapping V_i and V_{i+1}.

- If i and $i+1$ belong to distinct cycles in of π_V then the number of cycles decreases by one,
- otherwise it increases by one.

N.B.: a generalization of a technique from (Giuliani, L., Masillo, Rizzi, 2021)

Lemma (Swap Lemma) Let V be a binary string, $V_i \neq V_{i+1}$, and V' the result of swapping V_i and V_{i+1}.

- If i and $i+1$ belong to distinct cycles in of π_V then the number of cycles decreases by one,
- otherwise it increases by one.

N.B.: a generalization of a technique from (Giuliani, L., Masillo, Rizzi, 2021)

Ex. $V = \underset{0\,1\,2\,3\,4\,5\,6\,7}{\texttt{abbababa}}$, then $\pi_V = (0)(1,4,6,7,3)(2,5)$.

Lemma (Swap Lemma) Let V be a binary string, $V_i \neq V_{i+1}$, and V' the result of swapping V_i and V_{i+1}.

- If i and $i+1$ belong to distinct cycles in of π_V then the number of cycles decreases by one,
- otherwise it increases by one.

N.B.: a generalization of a technique from (Giuliani, L., Masillo, Rizzi, 2021)

Ex. $V = \underset{0\,1\,2\,3\,4\,5\,6\,7}{\texttt{abbababa}}$, then $\pi_V = (0)(1,4,6,7,3)(2,5)$.

- swap V_0 and V_1 : `babababa`, st. perm. $(0,4,6,7,3,1)(2,5)$

Lemma (Swap Lemma) Let V be a binary string, $V_i \neq V_{i+1}$, and V' the result of swapping V_i and V_{i+1}.

- If i and $i+1$ belong to distinct cycles in of π_V then the number of cycles decreases by one,
- otherwise it increases by one.

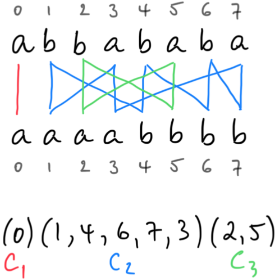
N.B.: a generalization of a technique from (Giuliani, L., Masillo, Rizzi, 2021)

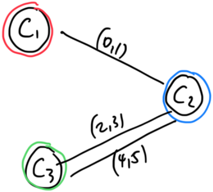
Ex. $V = \underset{0\,1\,2\,3\,4\,5\,6\,7}{\texttt{abbababa}}$, then $\pi_V = (0)(1,4,6,7,3)(2,5)$.

- swap V_0 and V_1 : `babababa`, st. perm. $(0,4,6,7,3,1)(2,5)$
- swap V_2 and V_3 : `baabbaba`, st. perm. $(0,4,6,7,3,5,2,1)$

Invert `baabbaba` and output the dB sequence $T =$ `aaababbb`.

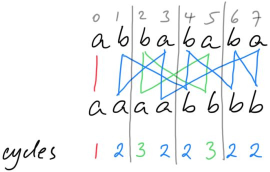
How to choose the blocks to swap

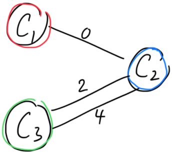


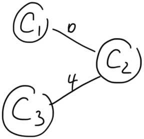


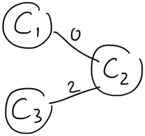
- unhappy block: elements $2i, 2i+1$ are in different cycles
- cycle graph Γ_V: vertices = cycles, edges = unhappy blocks
- Spanning Trees of Γ_V = (BWTs of) dB sequences closest to V
- here 2 STs: BWTs of `aaabbbab`, `aaababbb`

example cont.
here 2 STs:









babaabba

aaabbbab

baabbaba

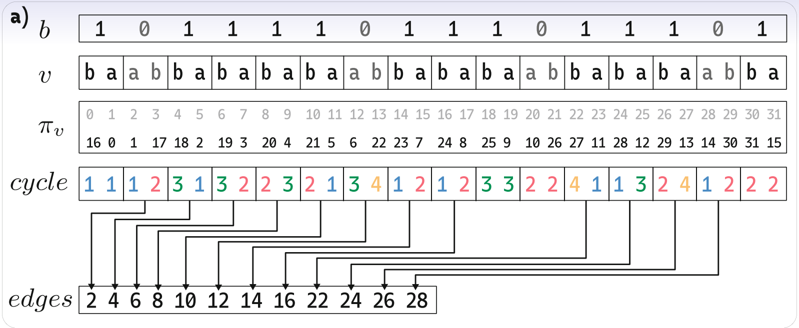
aaababbb

Some final details

- The standard permutation can be computed easily: the *i*th 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` ↦ 0, `ba` ↦ 1.
- enc(`babaabba`) = 1101, dec(1101) = `babaabba`

Algorithm overview (conceptual)

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 $\mathcal{T}$ of Γ_V.
5. Flip the bits of b corresponding to $\mathcal{T}$, resulting in b'.
6. Invert $S = dec(b')$, resulting in dB sequence T.



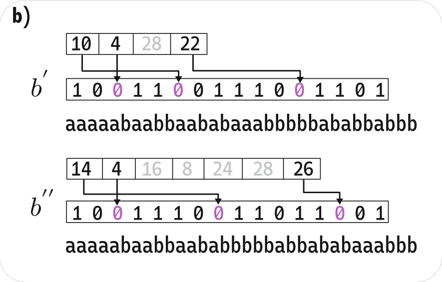
a)

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

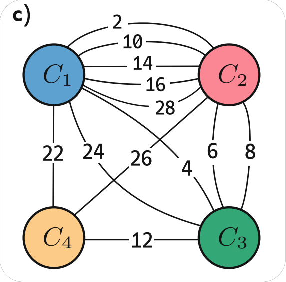
b': 1 0 0 1 1 0 0 1 1 1 0 0 1 1 0 1

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



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 $\mathcal{T}$ 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 $\mathcal{T}$ 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. (in parallel with 4.:) Flip the bits of b of $\mathcal{T}$, resulting in b'. $\mathcal{O}(n)$

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 $\mathcal{T}$ 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. (in parallel with 4.:) Flip the bits of b of $\mathcal{T}$, resulting in b'. $\mathcal{O}(n)$
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 $\mathcal{T}$ 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. (in parallel with 4.:) Flip the bits of b of $\mathcal{T}$, resulting in b'. $\mathcal{O}(n)$
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.

BWT-based algorithm for generating random dB sequences

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

BWT-based algorithm for generating random dB sequences

- first practical algorithm for constructing a random dB sequence which can produce every 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)

BWT-based algorithm for generating random dB sequences

- first practical algorithm for constructing a random dB sequence which can produce every 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)
- or just try it online: `debruijnsequence.org/db/random`

BWT-based algorithm for generating random dB sequences

- first practical algorithm for constructing a random dB sequence which can produce every 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)
- or just try it online: `debruijnsequence.org/db/random`
- can be straighforwardly extended to any constant-size alphabet (present on our github)

Conclusion and open problems

Conclusion: The BWT is good for more than compression and indexing!

Conclusion and open problems

Conclusion: The BWT is good for more than compression and indexing!

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:** Zs. Lipták and L. Parmigiani: *A BWT-based algorithm for random de Bruijn sequence construction*, LATIN 2024.
- **Implementation:** `github.com/lucaparmigiani/rnd_dbseq`

- **Paper:** Zs. Lipták and L. Parmigiani: *A BWT-based algorithm for random de Bruijn sequence construction*, LATIN 2024.
- **Implementation:** `github.com/lucaparmigiani/rnd_dbseq`

Thank you for your attention!

`zsuzsanna.liptak@univr.it`

Appendix

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

k-mer	rotation	power
aaa	aaa	$\mathtt{a}^3$
aab	aabbb	aabbb
aba	abab	$(\mathtt{ab})^2$
abb	abbba	aabbb
baa	...	
bab		
bba		
bbb		

Thm (Higgins, 2012) The set $\{\mathtt{ab}, \mathtt{ba}\}^{2^{k-1}}$ is the set of eBWTs of binary de Bruijn sets of order k.

A case study

Estimating the average discrepancy of de Bruijn sequences

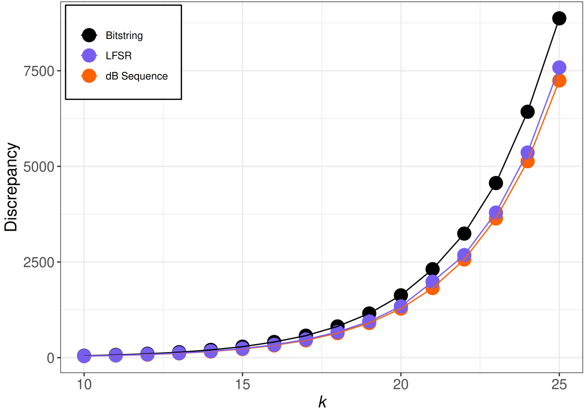
Def. The discrepancy of a binary string is the maximum absolute difference between the number of a's and b's over all (circular) substrings.

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

AAAAAABAAAABBAAABABAAABBBAABAABABBAABBABAABBBBABABABBBABBABBBBBB

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

Estimating the average discrepancy of dB sequences

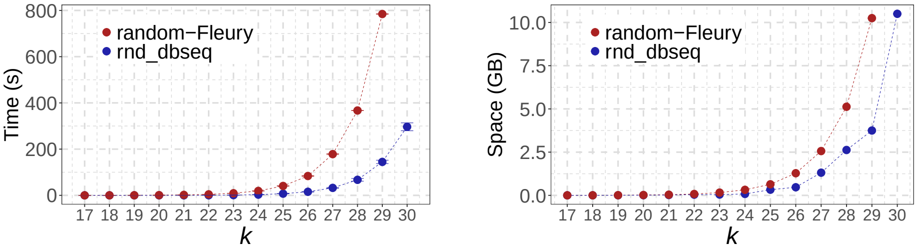


Average discrepancy of LFSRs from (Gabric and Sawada, 2022).

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

Comparison with a randomized 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.

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

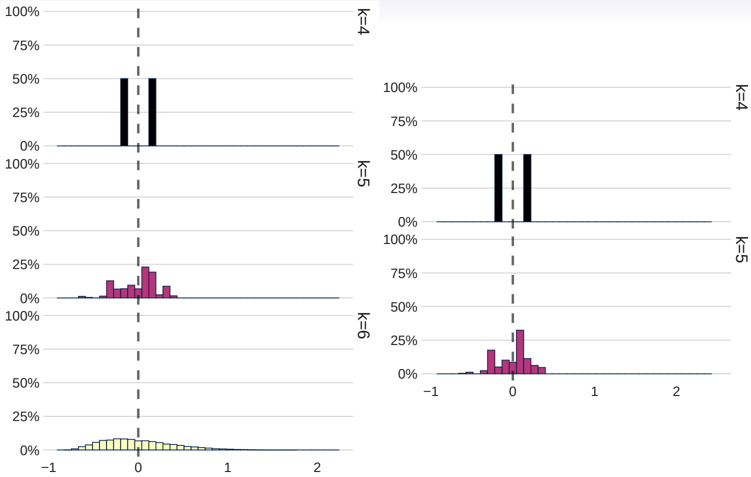
Our algorithm does not output all dB sequences according to the uniform probability distribution, for two reasons:

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

$$pres(T) = \frac{1}{2^{2^{k-1}}} \sum_{V \in \{\mathsf{ab},\mathsf{ba}\}^{2^{k-1}}} prob(T \mid V)$$



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.