Dollar or no dollar, that is the question

New combinatorial results on the Burrows-Wheeler-Transform

Zsuzsanna Lipták

University of Verona (Italy)

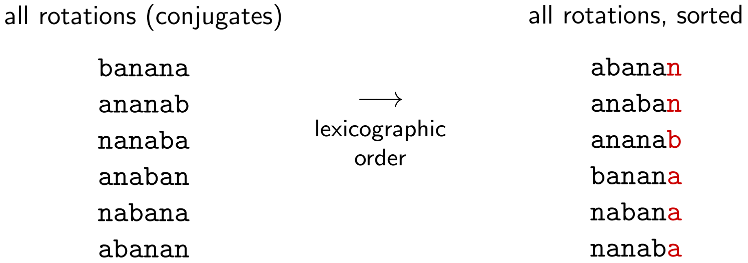
18e Journées Montoises d'Informatique Théorique
Prague, 9 Sept. 2022

Part I:

Introduction

The Burrows-Wheeler-Transform

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



all rotations (conjugates)

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

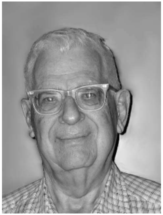
$\longrightarrow$ lexicographic order

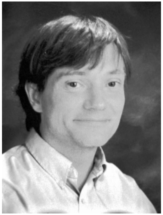
all rotations, sorted

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

Take a string (word) T, list all of its rotations, sort them lexicographically, concatenate last characters: bwt(T).

BWT history

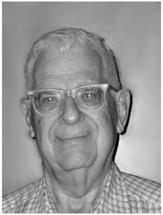


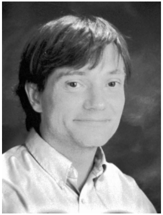


- invented by David Wheeler in the 70s
  as a lossless text compression algorithm
- fully developed and written up together with Michael Burrows in 1994
- appeared as a technical report only, never published
- popularized by Julian Seward's implementation: `bzip` and `bzip2` (1996)

source: Adjeroh, Bell, Mukerjee: The Burrows-Wheeler-Transform, Springer, 2008

BWT history





- invented by David Wheeler in the 70s
  as a **lossless** text compression algorithm
- fully developed and written up together with Michael Burrows in 1994
- appeared as a technical report only, never published
- popularized by Julian Seward's implementation: `bzip` and `bzip2` (1996)

source: Adjeroh, Bell, Mukerjee: The Burrows-Wheeler-Transform, Springer, 2008

Reversing the BWT

input: nnbaaa, 3 bwt(T), i: where $0 \leq i < n$

output: (wanted) banana. T: i'th rotation lex.ly

Reversing the BWT

input: nnbaaa, 3 bwt(T), i: where $0 \leq i < n$

output: (wanted) banana. T: i'th rotation lex.ly

Recall: BWT-matrix (F: first column, L: last column)

$$\begin{array}{cccccccc}
 & \text{F} & & & & & \text{L} \\
0 & \texttt{a} & \texttt{b} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{n} \\
1 & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{b} & \texttt{a} & \texttt{n} \\
2 & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{b} \\
3 & \texttt{b} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{n} & \texttt{a} \\
4 & \texttt{n} & \texttt{a} & \texttt{b} & \texttt{a} & \texttt{n} & \texttt{a} \\
5 & \texttt{n} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{b} & \texttt{a}
\end{array}$$

Reversing the BWT

input: nnbaaa, 3 bwt(T), i: where $0 \leq i < n$

output: (wanted) banana. T: i'th rotation lex.ly

Recall: BWT-matrix (F: first column, L: last column)

	F		L
0	a	bana	n
1	a	naba	n
2	a	nana	b
3	b	anan	a
4	n	aban	a
5	n	anab	a

- **Obs. 1:** F = all characters of T in lex. order: aaabnn

Reversing the BWT

input: `nnbaaa`, 3 bwt(T), i: where $0 \leq i < n$
output: (wanted) `banana`. T: i'th rotation lex.ly

Recall: BWT-matrix (F: first column, L: last column)

```
    F    L
0   abanan
1   anaban
2   ananab
3   banana
4   nabana
5   nanaba
```

- **Obs. 1:** F = all characters of T in lex. order: `aaabnn`
- **Obs. 2:** for all i: L_i precedes F_i in T:

  $T = \underset{0}{\texttt{b}}\underset{1}{\texttt{a}}\underset{2}{\texttt{n}}\underset{3}{\texttt{a}}\underset{4}{\texttt{n}}\underset{5}{\texttt{a}}$

Reversing the BWT

input: nnbaaa, 3 — bwt(T), i: where $0 \leq i < n$

output: (wanted) banana. — T: i'th rotation lex.ly

Recall: BWT-matrix (F: first column, L: last column)

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

- **Obs. 1:** F = all characters of T in lex. order: aaabnn
- **Obs. 2:** for all i: L_i precedes F_i in T:

  $T = \underset{0}{\text{b}}\underset{1}{\text{a}}\underset{2}{\text{n}}\underset{3}{\text{a}}\underset{4}{\text{n}}\underset{5}{\text{a}}$

Thm. (LF-property): The j'th occurrence of character x in L is the j'th occurrence of character x in F.

Reversing the BWT

- **Obs. 1:** F = all characters of T in lex. order
- **Obs. 2:** L_i precedes F_i in T
- **LF-property:** The j'th $\mathtt{x}$ in L is the j'th $\mathtt{x}$ in F.

Reversing the BWT

- **Obs. 1:** F = all characters of T in lex. order
- **Obs. 2:** L_i precedes F_i in T
- **LF-property:** The j'th `x` in L is the j'th `x` in F.

input: `nnbaaa`, 3

Reversing the BWT

- **Obs. 1:** F = all characters of T in lex. order
- **Obs. 2:** L_i precedes F_i in T
- **LF-property:** The j'th `x` in L is the j'th `x` in F.

input: `nnbaaa`, 3

	F		L
0	a	bana	n
1	a	naba	n
2	a	nana	b
3	b	anan	a
4	n	aban	a
5	n	anab	a

Reversing the BWT

- **Obs. 1:** F = all characters of T in lex. order
- **Obs. 2:** L_i precedes F_i in T
- **LF-property:** The j'th `x` in L is the j'th `x` in F.

input: `nnbaaa`, 3

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

Reversing the BWT

- **Obs. 1:** F = all characters of T in lex. order
- **Obs. 2:** L_i precedes F_i in T
- **LF-property:** The j'th `x` in L is the j'th `x` in F.

input: `nnbaaa`, 3

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

a

Reversing the BWT

- **Obs. 1:** F = all characters of T in lex. order
- **Obs. 2:** L_i precedes F_i in T
- **LF-property:** The j'th `x` in L is the j'th `x` in F.

input: `nnbaaa`, 3

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

n a

Reversing the BWT

- **Obs. 1:** F = all characters of T in lex. order
- **Obs. 2:** L_i precedes F_i in T
- **LF-property:** The j'th x in L is the j'th x in F.

input: nnbaaa, 3

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

a n a

Reversing the BWT

- **Obs. 1:** F = all characters of T in lex. order
- **Obs. 2:** L_i precedes F_i in T
- **LF-property:** The j'th x in L is the j'th x in F.

input: nnbaaa, 3

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

n a n a

Reversing the BWT

- **Obs. 1:** F = all characters of T in lex. order
- **Obs. 2:** L_i precedes F_i in T
- **LF-property:** The j'th `x` in L is the j'th `x` in F.

input: `nnbaaa`, 3

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

a n a n a

Reversing the BWT

- **Obs. 1:** F = all characters of T in lex. order
- **Obs. 2:** L_i precedes F_i in T
- **LF-property:** The j'th `x` in L is the j'th `x` in F.

input: `nnbaaa`, 3

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

banana

Why can the BWT be useful in text compression?

- **Obs. 2:** L_i precedes F_i in T

Why can the BWT be useful in text compression?

- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows

Why can the BWT be useful in text compression?

- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows

Ex.: $T =$ `banana` has 2 occurrences of the pattern `ana`

2 occ's of `ana`

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

Why can the BWT be useful in text compression?

- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows

Ex.: $T =$ `banana` has 2 occurrences of the pattern `ana`

2 occ's of `ana`	2 occ's of `na` preceded by `a`
`abanan`	`abanan`
`anaban`	`anaban`
`ananab`	`ananab`
`banana`	`banana`
`nabana`	`nabana`
`nanaba`	`nanaba`

Why can the BWT be useful in text compression?

- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows

Ex.: $T =$ `banana` has 2 occurrences of the pattern `ana`

2 occ's of `ana`	2 occ's of `na` preceded by `a`	2 occ's of `a` preceded by `n`
abanan	abanan	abanan
anaban	anaban	anaban
ananab	ananab	ananab
banana	banana	banana
nabana	nabana	nabana
nanaba	nanaba	nanaba

Why can the BWT be useful in text compression?

- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows

Ex.: $T =$ banana has 2 occurrences of the pattern ana

2 occ's of ana	2 occ's of na preceded by a	2 occ's of a preceded by n
abanan	abanan	abanan
anaban	anaban	anaban
ananab	ananab	ananab
banana	banana	banana
nabana	nabana	nabana
nanaba	nanaba	nanaba

So: we get a run of a's of length 2, and a run of n's of length 2

Why can the BWT be useful in text compression?

- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows

Ex.: $T =$ banana has 2 occurrences of the pattern ana

2 occ's of ana	2 occ's of na preceded by a	2 occ's of a preceded by n
abanan	abanan	abanan
anaban	anaban	anaban
ananab	ananab	ananab
banana	banana	banana
nabana	nabana	nabana
nanaba	nanaba	nanaba

So: we get a run of a's of length 2, and a run of n's of length 2 ($2 =$ no. occ's).

Of course, things are a bit more complicated:

Of course, things are a bit more complicated:

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

- many `the`’s, some `he, she, The`

Compression with the BWT

- in original paper: using Move-to-front and Huffman/arithmetic coding

Compression with the BWT

- in original paper: using Move-to-front and Huffman/arithmetic coding
- nowadays: using RLE (runlength-encoding)   We will soon see why!

Compression with the BWT

- in original paper: using Move-to-front and Huffman/arithmetic coding
- nowadays: using RLE (runlength-encoding) We will soon see why!
  - RLE: replace equal-letter-runs by (character, integer)-pair
  - Ex.: $\texttt{bbbbbbbbcaaaaaaaaaaabb} \mapsto (\texttt{b}, 8), (\texttt{c}, 1), (\texttt{a}, 11), (\texttt{b}, 2)$

Compression with the BWT

- in original paper: using Move-to-front and Huffman/arithmetic coding
- nowadays: using RLE (runlength-encoding) We will soon see why!
  - RLE: replace equal-letter-runs by (character, integer)-pair
  - Ex.: `bbbbbbbbcaaaaaaaaaaabb` $\mapsto (\texttt{b}, 8), (\texttt{c}, 1), (\texttt{a}, 11), (\texttt{b}, 2)$
- good if few runs w.r.t. length of string

Compression with the BWT

- in original paper: using Move-to-front and Huffman/arithmetic coding
- nowadays: using RLE (runlength-encoding) We will soon see why!
  - RLE: replace equal-letter-runs by (character, integer)-pair
  - Ex.: $\texttt{bbbbbbbbcaaaaaaaaaaabb} \mapsto (\texttt{b}, 8), (\texttt{c}, 1), (\texttt{a}, 11), (\texttt{b}, 2)$
- good if few runs w.r.t. length of string
- Def.: $r(T) = \#$ runs of $\mathrm{bwt}(T)$
  Ex.: $r(\texttt{banana}) = 3$ recall: $\mathrm{bwt}(\texttt{banana}) = \texttt{nnbaaa}$

Compression with the BWT

- in original paper: using Move-to-front and Huffman/arithmetic coding
- nowadays: using RLE (runlength-encoding) We will soon see why!
  - RLE: replace equal-letter-runs by (character, integer)-pair
  - Ex.: bbbbbbbbcaaaaaaaaaaabb $\mapsto (\texttt{b}, 8), (\texttt{c}, 1), (\texttt{a}, 11), (\texttt{b}, 2)$
- good if few runs w.r.t. length of string
- Def.: $r(T) = \#$ runs of bwt(T)
  Ex.: $r(\texttt{banana}) = 3$ recall: bwt(banana) = nnbaaa
- for repetitive strings, r is small (more on this later)

Pattern matching with the BWT

Most fundamental algorithmic problem on strings:

Pattern matching with the BWT

Most fundamental algorithmic problem on strings:

Pattern matching:
Given a string T of length n (the text) and a string P of length m (the pattern), find all occurrences of P in T as a substring.
Typically: $m << n$.

Pattern matching with the BWT

Most fundamental algorithmic problem on strings:

Pattern matching:
Given a string T of length n (the text) and a string P of length m (the pattern), find all occurrences of P in T as a substring.
Typically: $m << n$.
Variants: decide if P occurs, return the number of occurrences, find one occurrence, find first occurrence, ...

Pattern matching with the BWT

Most fundamental algorithmic problem on strings:

Pattern matching:
Given a string T of length n (the text) and a string P of length m (the pattern), find all occurrences of P in T as a substring.
Typically: $m << n$.
Variants: decide if P occurs, return the number of occurrences, find one occurrence, find first occurrence, ...

Ex.: $T = \underset{0\,1\,2\,3\,4\,5}{\texttt{banana}}$ and $P = \texttt{ana}$. $Occ(P) = \{1, 3\}$.

Pattern matching with the BWT

Most fundamental algorithmic problem on strings:

Pattern matching:
Given a string T of length n (the text) and a string P of length m (the pattern), find all occurrences of P in T as a substring.
Typically: $m << n$.
Variants: decide if P occurs, return the number of occurrences, find one occurrence, find first occurrence, ...

Ex.: $T = \underset{0\,1\,2\,3\,4\,5}{\texttt{banana}}$ and $P = \texttt{ana}$. $Occ(P) = \{1, 3\}$.

- without additional data structures, time $\Omega(n + m)$ (read the input)
- exist algorithms achieving $\Theta(n + m)$ worst-case (Knuth-Morris-Pratt)

Pattern matching with the BWT

Backward search [Ferragina and Manzini, 2000]

1. process pattern back-to-front
2. $Occ(xU) \subseteq Occ(U) - 1$

$Occ(U) =$ occurrences of U in T

Pattern matching with the BWT

Backward search [Ferragina and Manzini, 2000]

1. process pattern back-to-front
2. $Occ(xU) \subseteq Occ(U) - 1$

$Occ(U)$ = occurrences of U in T

ex. $T = \underset{0\,1\,2\,3\,4\,5}{\texttt{banana}}$ and $P = \texttt{ana}$.

$(Occ(\texttt{a}) = \{1, 3, 5\}, Occ(\texttt{na}) = \{2, 4\}, Occ(\texttt{ana}) = \{1, 3\})$.

all occ's of a	all occ's of na	all occ's of ana
abanan	abanan	abanan
anaban	anaban	anaban
ananab	ananab	ananab
banana	banana	banana
nabana	nabana	nabana
nanaba	nanaba	nanaba

Pattern matching with the BWT

Magic! Backward search can be done on the BWT directly (with some additional magic. . .):

Ex.: ~~T = banana~~ and P = ana.

bwt(T) = nnbaaa.

all occ's of a	all occ's of na	all occ's of ana
n	n	n
n	n	n
b	b	b
a	a	a
a	a	a
a	a	a

Pattern matching with the BWT

Magic! Backward search can be done on the BWT directly (with some additional magic. . .):

Ex.: ~~T = banana~~ and P = ana.

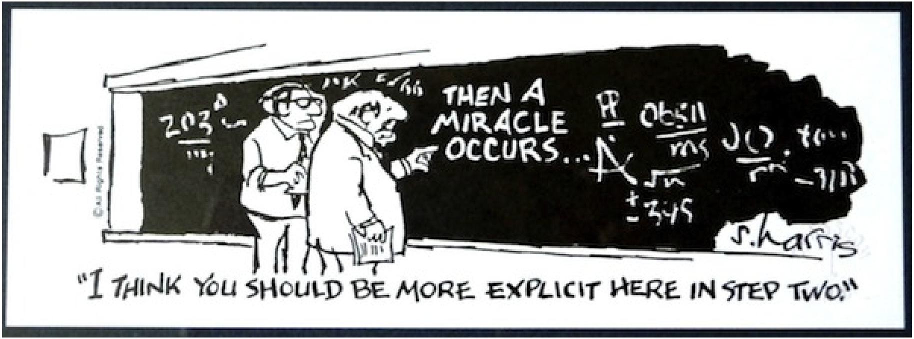
bwt(T) = nnbaaa.

all occ's of a	all occ's of na	all occ's of ana
n	n	n
n	n	n
b	b	b
a	a	a
a	a	a
a	a	a

Thm. Pattern matching on bwt(T) (decision and counting) can be implemented in $O(m \log \sigma)$ time, using only $o(n)$ additional bits.

σ = alphabetsize

BWT magic



copyright: Sydney Harris

BWT magic

The BWT ...

- requires same space as T in bits: $n \log \sigma$ bits $\sigma =$ alphabetsize
  (suffix array: $n \log n$ bits, suffix tree: much more — still $\mathcal{O}(n)$)

BWT magic

The BWT ...

- requires same space as T in bits: $n \log \sigma$ bits $\sigma =$ alphabetsize
  (suffix array: $n \log n$ bits, suffix tree: much more — still $\mathcal{O}(n)$)

We have seen:

- lossless: BWT is reversible: `nnbaaa`,3 ↦ `banana`
- easier to compress than T, if T repetitive
- pattern matching in $\mathcal{O}(m \log \sigma)$ time $m = |P|$
  (on T : $\mathcal{O}(n + m)$ time) $n = |T|$

BWT magic

The BWT ...

- requires same space as T in bits: $n \log \sigma$ bits $\sigma =$ alphabetsize
  (suffix array: $n \log n$ bits, suffix tree: much more — still $\mathcal{O}(n)$)

We have seen:

- lossless: BWT is reversible: `nnbaaa`,3 ↦ `banana`
- easier to compress than T, if T repetitive
- pattern matching in $\mathcal{O}(m \log \sigma)$ time $m = |P|$
  (on T : $\mathcal{O}(n + m)$ time) $n = |T|$

We have not seen:

- reversible in linear time $\mathcal{O}(n)$ $n = |T|$
- computable in linear time $\mathcal{O}(n)$
- can replace text (suffix array, suffix tree: no)

Compressed data structures for strings

The amount of (just HTML) online text material in the Web was estimated, in 2002, to exceed by 30-40 times what had been printed during the whole history of mankind.

from: Navarro & Mäkinen,
Compressed Full Text Indexes,

ACM Computing Surveys, 2007

Compressed data structures for strings

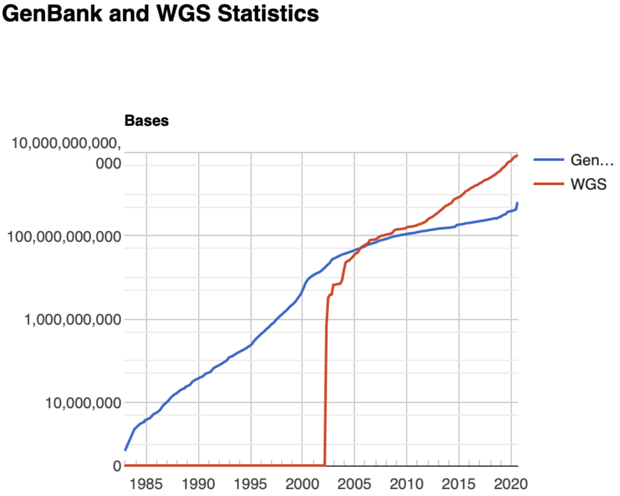
The amount of (just HTML) online text material in the Web was estimated, in 2002, to exceed by 30-40 times what had been printed during the whole history of mankind.

from: Navarro & Mäkinen,
Compressed Full Text Indexes,

ACM Computing Surveys, 2007

N.B. And this was in 2002!

Let's look at biological sequences . . .



GenBank and WGS Statistics

source: NCBI website

Compressed data structures for strings

So we need efficient ways of . . .

- storing,
- querying,
- mining,
- searching,
- . . .

. . . very large amounts of textual data.

Compressed data structures for strings

Some data structures based on the BWT:

- FM-index [Ferragina and Manzini, FOCS 2000]
- RLFM-index [Mäkinen and Navarro, CPM 2005]
- *r*-index [Gagie et al, JACM 2020; Bannai et al. TCS 2020]
- some recent developments on *r*-index [Rossi et al. JCB 2022; Giuliani et al. SEA 2022; Cobas et al. CPM 2021; Boucher et al. SPIRE 2021]

Some tools in bioinformatics (aligners):

bwa [Durbin and Li, 2009]	ca. 41,000 cit.
bowtie [Langmead and Salzberg, 2010]	ca. 36,000 cit.
soap2 [Li et al., 2009]	
...	

The parameter r

Def. String T, $r =$ number of runs of bwt(T).

- size of data structures $\mathcal{O}(r)$
- algorithms' running time ideally a function of r (not of $n = |T|$)
- increasingly used as a repetitiveness measure of T
- some papers on r:
    - Manzini: "An analysis of the Burrows-Wheeler-Transform" [JACM 2001]
    - Kempa and Kociumaka: "Resolution of the Burrows-Wheeler Transform Conjecture" [FOCS 2020]
    - Navarro: "Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures" [ACM Comp. Surv., 2021]
    - Mantaci et al.: "Measuring the clustering effect of BWT via RLE" [TCS 2017]

BWT from a combinatorial perspective

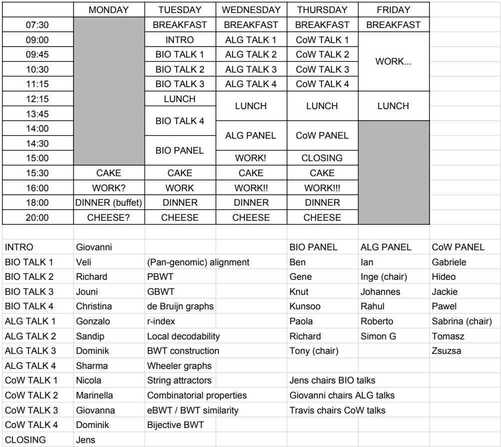
- special case of the Gessel-Reutenauer-bijection [Crochemore, Désarménien, Perrin, 2004]
- introduction of the extended BWT (eBWT), a generalization of the BWT to multisets of strings [Mantaci et al. 2007]
- strings T with fully clustering BWTs (e.g. $\text{bwt}(T) = \texttt{bbbbaaccc}$)
    - full characterization for $\sigma = 2$ [Mantaci et al., 2003]
    - partial characterization for $\sigma > 2$ [Puglisi et al., 2008]
    - characterization via interval exchanges [Ferenczi et al., 2013]
- fixpoints of the BWT [Mantaci et al., 2017]
- characterization of BWT images [Likhomanov and Shur, 2011]

Good overview: Rosone and Sciortino: "The Burrows-Wheeler Transform between Data Compression and Combinatorics on Words." [CiE 2013]

- two research communities working on the BWT
- (1) data structures and algorithms on strings and
  (2) combinatorics on words
- little interaction until . . .

Dagstuhl workshop "25 years of the Burrows-Wheeler-Transform" (2019) organized by T. Gagie, G. Manzini, G. Navarro, J. Stoye

The schedule:

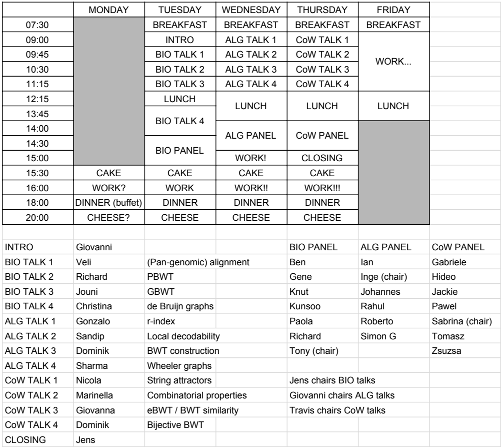


	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY
07:30		BREAKFAST	BREAKFAST	BREAKFAST	BREAKFAST
09:00		INTRO	ALG TALK 1	CoW TALK 1	WORK...
09:45		BIO TALK 1	ALG TALK 2	CoW TALK 2	
10:30		BIO TALK 2	ALG TALK 3	CoW TALK 3	
11:15		BIO TALK 3	ALG TALK 4	CoW TALK 4	
12:15		LUNCH	LUNCH	LUNCH	LUNCH
13:45		BIO TALK 4			
14:00			ALG PANEL	CoW PANEL	
14:30		BIO PANEL			
15:00			WORK!	CLOSING	
15:30	CAKE	CAKE	CAKE	CAKE	
16:00	WORK?	WORK	WORK!!	WORK!!!	
18:00	DINNER (buffet)	DINNER	DINNER	DINNER	
20:00	CHEESE?	CHEESE	CHEESE	CHEESE	

INTRO	Giovanni			BIO PANEL	ALG PANEL	CoW PANEL
BIO TALK 1	Veli	(Pan-genomic) alignment		Ben	Ian	Gabriele
BIO TALK 2	Richard	PBWT		Gene	Inge (chair)	Hideo
BIO TALK 3	Jouni	GBWT		Knut	Johannes	Jackie
BIO TALK 4	Christina	de Bruijn graphs		Kunsoo	Rahul	Pawel
ALG TALK 1	Gonzalo	r-index		Paola	Roberto	Sabrina (chair)
ALG TALK 2	Sandip	Local decodability		Richard	Simon G	Tomasz
ALG TALK 3	Dominik	BWT construction		Tony (chair)		Zsuzsa
ALG TALK 4	Sharma	Wheeler graphs				
CoW TALK 1	Nicola	String attractors		Jens chairs BIO talks		
CoW TALK 2	Marinella	Combinatorial properties		Giovanni chairs ALG talks		
CoW TALK 3	Giovanna	eBWT / BWT similarity		Travis chairs CoW talks		
CoW TALK 4	Dominik	Bijective BWT				
CLOSING	Jens					

At the workshop, the communities were called

The schedule:



	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY
07:30		BREAKFAST	BREAKFAST	BREAKFAST	BREAKFAST
09:00		INTRO	ALG TALK 1	CoW TALK 1	WORK...
09:45		BIO TALK 1	ALG TALK 2	CoW TALK 2	
10:30		BIO TALK 2	ALG TALK 3	CoW TALK 3	
11:15		BIO TALK 3	ALG TALK 4	CoW TALK 4	
12:15		LUNCH	LUNCH	LUNCH	LUNCH
13:45		BIO TALK 4			
14:00			ALG PANEL	CoW PANEL	
14:30		BIO PANEL			
15:00			WORK!	CLOSING	
15:30	CAKE	CAKE	CAKE	CAKE	
16:00	WORK?	WORK	WORK!!	WORK!!!	
18:00	DINNER (buffet)	DINNER	DINNER	DINNER	
20:00	CHEESE?	CHEESE	CHEESE	CHEESE	

INTRO	Giovanni		BIO PANEL	ALG PANEL	CoW PANEL
BIO TALK 1	Veli	(Pan-genomic) alignment	Ben	Ian	Gabriele
BIO TALK 2	Richard	PBWT	Gene	Inge (chair)	Hideo
BIO TALK 3	Jouni	GBWT	Knut	Johannes	Jackie
BIO TALK 4	Christina	de Bruijn graphs	Kunsoo	Rahul	Pawel
ALG TALK 1	Gonzalo	r-index	Paola	Roberto	Sabrina (chair)
ALG TALK 2	Sandip	Local decodability	Richard	Simon G	Tomasz
ALG TALK 3	Dominik	BWT construction	Tony (chair)		Zsuzsa
ALG TALK 4	Sharma	Wheeler graphs			
CoW TALK 1	Nicola	String attractors	Jens chairs BIO talks		
CoW TALK 2	Marinella	Combinatorial properties	Giovanni chairs ALG talks		
CoW TALK 3	Giovanna	eBWT / BWT similarity	Travis chairs CoW talks		
CoW TALK 4	Dominik	Bijective BWT			
CLOSING	Jens				

At the workshop, the communities were called ALG, BIO, and CoW (sic!)

But: The two communities use slightly different definitions of the BWT:

- ALG (incl. BIO): It is assumed that each string terminates with an end-of-string character (denoted \$, smaller than all others) $T = \texttt{banana\$}$
- CoW: no such assumption $T = \texttt{banana}$

But: The two communities use slightly different definitions of the BWT:

- ALG (incl. BIO): It is assumed that each string terminates with an end-of-string character (denoted \$, smaller than all others)
  $T = \texttt{banana\$}$
- CoW: no such assumption $T = \texttt{banana}$

This talk is about the implications of this difference.

Part II:

Dollar or no dollar, that is the question

- ALG (incl. BIO): It is assumed that each string terminates with an end-of-string character (denoted \$) $T = \texttt{banana\$}$
- CoW: no such assumption $T = \texttt{banana}$

- ALG (incl. BIO): It is assumed that each string terminates with an end-of-string character (denoted \$) $T = \texttt{banana\$}$
- CoW: no such assumption $T = \texttt{banana}$

This talk is about the implications of this difference.

In particular:

- ALG (incl. BIO): It is assumed that each string terminates with an end-of-string character (denoted \$) $T = \texttt{banana\$}$
- CoW: no such assumption $T = \texttt{banana}$

This talk is about the implications of this difference.

In particular:

1. the transform itself

- ALG (incl. BIO): It is assumed that each string terminates with an end-of-string character (denoted \$) $T = \texttt{banana\$}$
- CoW: no such assumption $T = \texttt{banana}$

This talk is about the implications of this difference.

In particular:

1. the transform itself
2. BWT construction

- ALG (incl. BIO): It is assumed that each string terminates with an end-of-string character (denoted \$) $T =$ `banana$`
- CoW: no such assumption $T =$ `banana`

This talk is about the implications of this difference.

In particular:

1. the transform itself
2. BWT construction
3. BWT images

- ALG (incl. BIO): It is assumed that each string terminates with an end-of-string character (denoted \$) $T =$ `banana$`
- CoW: no such assumption $T =$ `banana`

This talk is about the implications of this difference.

In particular:

1. the transform itself
2. BWT construction
3. BWT images
4. BWT of string collections

1. The transform itself

Different transforms

banana

abanan
anaban
ananab
banana
nabana
nanaba

nnbaaa

banana$

$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba

annb$aa

Different transforms

	without dollar (banana)	with dollar (banana$)
the transform	nnbaaa	annb$aa

Different transforms

	without dollar (banana)	with dollar (banana$)
the transform	nnbaaa	annb$aa
remove $	nnbaaa	annbaa

Different transforms

	without dollar (banana)	with dollar (banana$)
the transform	nnbaaa	annb$aa
remove $	nnbaaa	annbaa
# runs r	3	4

Different transforms

	without dollar (banana)	with dollar (banana$)
the transform	nnbaaa	annb$aa
remove $	nnbaaa	annbaa
$\#$ runs r	3	4

- **Thm.** There exist strings for which the difference in r is $\Theta(\log n)$.

[Giuliani, Inenaga, L., Sciortino, 2022, forthcoming]

Different transforms

	without dollar (banana)	with dollar (banana$)
the transform	nnbaaa	annb$aa
remove $	nnbaaa	annbaa
# runs r	3	4

- **Thm.** There exist strings for which the difference in r is $\Theta(\log n)$. [Giuliani, Inenaga, L., Sciortino, 2022, forthcoming]
- This is asymptotically tight: here $r = O(1)$, and upper bound is $\mathcal{O}(\log r \log n)$. [Akagi, Funakoshi, Inenaga, 2021]

Different transforms

Thm. There exist strings for which the difference in r is $\Theta(\log n)$.

- $r(T\$)$ increases by $\log n$: Fibonacci words of even order $T = \mathit{Fib}(2k), r(T) = 2, r(T\$) = 2k - 1$

  ex.:
  $r(\mathit{Fib}(8)) = 2, r(\mathit{Fib}(8)\$) = 7$
  $r(\mathit{Fib}(12)) = 2, r(\mathit{Fib}(12)\$) = 11$

- $r(T\$)$ decreases by $\log n$: Fibonacci words of odd order without the first character $T = \mathit{Fib}(2k+1)[1:], r(T) = 2k, r(T\$) = 5$

  ex:
  $r(\mathit{Fib}(13)[1:]) = 12, r(\mathit{Fib}(13)[1:]\$) = 5$
  $r(\mathit{Fib}(15)[1:]) = 14, r(\mathit{Fib}(15)[1:]\$) = 5$

Different transforms

Thm. There exist strings for which the difference in r is $\Theta(\log n)$.

- $r(T\$)$ increases by $\log n$: Fibonacci words of even order $T = Fib(2k), r(T) = 2, r(T\$) = 2k - 1$

  ex.:
  $r(Fib(8)) = 2, r(Fib(8)\$) = 7$
  $r(Fib(12)) = 2, r(Fib(12)\$) = 11$

- $r(T\$)$ decreases by $\log n$: Fibonacci words of odd order without the first character $T = Fib(2k + 1)[1 :], r(T) = 2k, r(T\$) = 5$

  ex:
  $r(Fib(13)[1 :]) = 12, r(Fib(13)[1 :]\$) = 5$
  $r(Fib(15)[1 :]) = 14, r(Fib(15)[1 :]\$) = 5$

- both additive and multiplicative difference

2. BWT construction

BWT construction

Most BWT construction algorithms first construct the Suffix Array (SA), then construct the BWT from the SA, using: $L_i = T_{SA[i]-1}$ (recall Obs. 2).

ex. $T = \underset{0}{\texttt{b}}\underset{1}{\texttt{a}}\underset{2}{\texttt{n}}\underset{3}{\texttt{a}}\underset{4}{\texttt{n}}\underset{5}{\texttt{a}}\underset{6}{\texttt{\$}}$.

SA	
6	$
5	a$
3	ana$
1	anana$
0	banana$
4	na$
2	nana$

BWT construction

Most BWT construction algorithms first construct the Suffix Array (SA), then construct the BWT from the SA, using: $L_i = T_{SA[i]-1}$ (recall Obs. 2).

ex. $T = \underset{0}{\texttt{b}}\underset{1}{\texttt{a}}\underset{2}{\texttt{n}}\underset{3}{\texttt{a}}\underset{4}{\texttt{n}}\underset{5}{\texttt{a}}\underset{6}{\texttt{\$}}$.

SA	
6	$
5	a$
3	ana$
1	anana$
0	banana$
4	na$
2	nana$

SA		L
6	$banan	a
5	a$bana	n
3	ana$ba	n
1	anana$	b
0	banana	$
4	na$ban	a
2	nana$b	a

BWT construction

Most BWT construction algorithms first construct the Suffix Array (SA), then construct the BWT from the SA, using: $L_i = T_{SA[i]-1}$ (recall Obs. 2).

ex. $T = \underset{0\,1\,2\,3\,4\,5\,6}{\texttt{banana\$}}$.

SA	
6	$
5	a$
3	ana$
1	anana$
0	banana$
4	na$
2	nana$

SA	L
6	$banana
5	a$banan
3	ana$ban
1	anana$b
0	banana$
4	na$bana
2	nana$ba

Thus: SA-construction in $\mathcal{O}(n)$ time ⇒ BWT-construction in $\mathcal{O}(n)$ time.

BWT construction without dollar

- This works well if there is a \$.
- What if there is no dollar?

BWT construction without dollar

- This works well if there is a $.
- What if there is no dollar?

banana
0 1 2 3 4 5

SA	
5	a
3	ana
1	anana
0	banana
4	na
2	nana

BWT construction without dollar

- This works well if there is a $.
- What if there is no dollar?

banana
0 1 2 3 4 5

SA		SA	L
5	a	5	abanan
3	ana	3	anaban
1	anana	1	ananab
0	banana	0	banana
4	na	4	nabana
2	nana	2	nanaba

nnbaaa ✓

BWT construction without dollar

- This works well if there is a $.
- What if there is no dollar?

banana
0 1 2 3 4 5

anaban
0 1 2 3 4 5

SA		SA	L
5	a	5	abanan
3	ana	3	anaban
1	anana	1	ananab
0	banana	0	banana
4	na	4	nabana
2	nana	2	nanaba

nnbaaa ✓

BWT construction without dollar

- This works well if there is a $.
- What if there is no dollar?

banana
0 1 2 3 4 5

SA	
5	a
3	ana
1	anana
0	banana
4	na
2	nana

SA		L
5	abana	n
3	anaba	n
1	anana	b
0	banan	a
4	naban	a
2	nanab	a

nnbaaa ✓

anaban
0 1 2 3 4 5

SA	
2	aban
4	an
0	anaban
3	ban
5	n
1	naban

BWT construction without dollar

- This works well if there is a $.
- What if there is no dollar?

banana
0 1 2 3 4 5

SA		SA	L
5	a	5	abanan
3	ana	3	anaban
1	anana	1	ananab
0	banana	0	banana
4	na	4	nabana
2	nana	2	nanaba

nnbaaa ✓

anaban
0 1 2 3 4 5

SA		SA	L
2	aban	2	abanan
4	an	4	ananab
0	anaban	0	anaban
3	ban	3	banana
5	n	5	nabana
1	naban	1	nabana

nbnaaa ✗

BWT construction without dollar

- This works well if there is a $.
- What if there is no dollar?

banana
012345

SA		SA	L
5	a	5	abanan
3	ana	3	anaban
1	anana	1	ananab
0	banana	0	banana
4	na	4	nabana
2	nana	2	nanaba

nnbaaa ✓

anaban
012345

SA		SA	L
2	aban	2	abanan
4	an	4	ananab
0	anaban	0	anaban
3	ban	3	banana
5	n	5	nabana
1	naban	1	nabana

nbnaaa ✗

Problem 1: $suf_i < suf_j \Leftrightarrow conj_i < conj_j$ does not hold in general!

Thus: We need the CA (conjugate array), not the SA!

BWT construction without dollar

Problem 2: If T not primitive, then CA not defined (several identical rotations):

$\underset{0\,1\,2\,3\,4\,5}{\texttt{nanana}} = (\texttt{na})^3$

CA

```
1  ananan
3  ananan
5  ananan
0  nanana
2  nanana
4  nanana
```

Linear-time BWT construction without dollar

- For \$-terminated strings, neither problem exists.
- Same for Lyndon words (primitive and $<$ all their rotations).
- All previous BWT-construction algorithms either use \$ or Lyndon rotations.

Our algorithm [Boucher, Cenzato, L., Rossi, Sciortino, SPIRE, 2021]:

- first linear-time BWT-construction algorithm which uses neither \$ nor Lyndon rotations
- adaptation of the SAIS-algorithm for SA-construction [Nong et al., 2011]
- previously, SAIS had been adapted for $T\$$ [Okanohara and Sadakane 2009], and to the bijective BWT [Bannai et al., 2021]

Our algorithm for BWT construction

1. assign circular types to positions
2. sort LMS-substrings
3. assign new names to LMS-substrings
4. construct new string, solve recursively
5. induce CA from relative order of LMS-positions

Step 1

0	1	2	3	4	5
b	a	n	a	n	a
L	S	L	S	L	S
	*		*		*

Step 2

	a	*b*	*n*
S^*	1 3 5		
L		0	2 4
S	5 1 3		
	5 1 3	0	2 4

Step 3

5	a b a	A
1	a n a	B
3	a n a	B

Step 4

0	1	2
A	B	B
S	L	L
*		

A	*B*
0	
	2 1
0	2 1

Step 5

	a	*b*	*n*
	5 3 1		
		0	4 2
CA	5 3 1	0	4 2
BWT	n n b	a	a a

3. BWT images

BWT images

The BWT-mapping $\mathrm{bwt} : \Sigma^n \to \Sigma^n$, $T \mapsto \mathrm{bwt}(T)$ is not bijective:

- $\mathrm{bwt}(T) = \mathrm{bwt}(T') \iff T$ and T' are conjugates.
- Thus, not every word W is a BWT-image.
- Characterization of BWT-images exists (next)

BWT images

Idea: If a word *W* is a BWT-image, then it can be reversed:

	F	*L*
0	a	b
1	a	a
2	a	n
3	b	a
4	n	n
5	n	a

[1] a.k.a. standard permutation

BWT images

Idea: If a word *W* is a BWT-image, then it can be reversed:

	F	*L*
0	a	b
1	a	a
2	a	n
3	b	a
4	n	n
5	n	a

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

[1] a.k.a. standard permutation

BWT images

Idea: If a word *W* is a BWT-image, then it can be reversed:

	F	*L*
0	a	b
1	a	a
2	a	n
3	b	a
4	n	n
5	n	a

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

We get: aab, of length $< n = 6$. ✗

[1] a.k.a. standard permutation

BWT images

Idea: If a word W is a BWT-image, then it can be reversed:

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

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

We get: aab, of length $< n = 6$. ✗

In other words, the permutation defined by the LF-mapping[1] has more than one cycle: $\left(\begin{smallmatrix}0&1&2&3&4&5\\3&0&4&1&5&2\end{smallmatrix}\right) = (0,3,1)(2,4,5)$.

[1] a.k.a. standard permutation

BWT images

Def. Given a word W, its standard permutation π is defined by: $\pi(i) < \pi(j)$ iff (a) $W[i] < W[j]$ or (b) $W[i] = W[j]$ and $i < j$.

BWT images

Def. Given a word W, its standard permutation π is defined by:
$\pi(i) < \pi(j)$ iff (a) $W[i] < W[j]$ or (b) $W[i] = W[j]$ and $i < j$.

Thm. [Likhomanov and Shur, 2011] A word W is the BWT of some word iff the number of cycles of its standard permutation π equals the gcd of its runlengths.

BWT images

Def. Given a word W, its standard permutation π is defined by:
$\pi(i) < \pi(j)$ iff (a) $W[i] < W[j]$ or (b) $W[i] = W[j]$ and $i < j$.

Thm. [Likhomanov and Shur, 2011] A word W is the BWT of some word iff the number of cycles of its standard permutation π equals the gcd of its runlengths.

Ex. `banana`, runlengths: 1,1,1,1,1,1, gcd $= 1$, π has 2 cycles: ✗

BWT images

Def. Given a word W, its standard permutation π is defined by: $\pi(i) < \pi(j)$ iff (a) $W[i] < W[j]$ or (b) $W[i] = W[j]$ and $i < j$.

Thm. [Likhomanov and Shur, 2011] A word W is the BWT of some word iff the number of cycles of its standard permutation π equals the gcd of its runlengths.

Ex. `banana`, runlengths: 1,1,1,1,1,1, gcd $= 1$, π has 2 cycles: ✗

Ex. `nnbaaa`, runlengths: 2,1,3, gcd $= 1$,
$\pi = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 3 & 0 & 1 & 2 \end{smallmatrix}\right) = (0, 4, 1, 5, 2, 3)$ has 1 cycle: ✓ bwt(`banana`)

BWT images

Def. Given a word W, its standard permutation π is defined by:
$\pi(i) < \pi(j)$ iff (a) $W[i] < W[j]$ or (b) $W[i] = W[j]$ and $i < j$.

Thm. [Likhomanov and Shur, 2011] A word W is the BWT of some word iff the number of cycles of its standard permutation π equals the gcd of its runlengths.

Ex. `banana`, runlengths: 1,1,1,1,1,1, gcd $= 1$, π has 2 cycles: ✗

Ex. `nnbaaa`, runlengths: 2,1,3, gcd $= 1$,
$\pi = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 3 & 0 & 1 & 2 \end{smallmatrix}\right) = (0,4,1,5,2,3)$ has 1 cycle: ✓ bwt(`banana`)

Ex. `nnnaaa`, runlengths: 3,3, gcd $= 3$,
$\pi = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 0 & 1 & 2 \end{smallmatrix}\right) = (0,3)(1,4)(3,5)$ 3 cycles: ✓ bwt(`ananan`)

BWT images with dollar

And with dollar?

- W has exactly one occurrence of \$ $\Longrightarrow$ $\text{gcd} = 1$.
- Thm. of Likhomanov and Shur: W is a BWT-image iff π is cyclic.
- Note that W has at most one pre-image (\$ is at the end).

When a dollar makes a BWT

[Giuliani, L., Masillo, Rizzi, TCS, 2021]

But we can ask a more complex question now:

Let $\mathrm{bwt}_{\$} : \Sigma^n \to \Sigma^n, T \mapsto \mathrm{bwt}(T\$)$ without the dollar.

ex. `banana` ↦ `annbaa`, since bwt(`banana$`) = `annb$aa`.

When a dollar makes a BWT

[Giuliani, L., Masillo, Rizzi, TCS, 2021]

But we can ask a more complex question now:

Let $\mathrm{bwt}_{\$} : \Sigma^n \to \Sigma^n, T \mapsto \mathrm{bwt}(T\$)$ without the dollar.

ex. banana ↦ annbaa, since bwt(banana$) = annb$aa.

Questions:

- Is $\mathrm{bwt}_{\$}$ bijective? (no)

When a dollar makes a BWT

[Giuliani, L., Masillo, Rizzi, TCS, 2021]

But we can ask a more complex question now:

Let $\mathrm{bwt}_\$: \Sigma^n \to \Sigma^n$, $T \mapsto \mathrm{bwt}(T\$)$ without the dollar.

ex. `banana` ↦ `annbaa`, since bwt(`banana$`) = `annb$aa`.

Questions:

- Is $\mathrm{bwt}_\$$ bijective? (no)
- Can we characterize $\mathrm{bwt}_\$$-images?

When a dollar makes a BWT

[Giuliani, L., Masillo, Rizzi, TCS, 2021]

But we can ask a more complex question now:

Let $\text{bwt}_\$: \Sigma^n \to \Sigma^n$, $T \mapsto \text{bwt}(T\$)$ without the dollar.

ex. `banana` ↦ `annbaa`, since bwt(`banana$`) = `annb$aa`.

Questions:

- Is $\text{bwt}_\$$ bijective? (no)
- Can we characterize $\text{bwt}_\$$-images?
- If W is a $\text{bwt}_\$$-image, how many distinct T's map to it?

When a dollar makes a BWT

[Giuliani, L., Masillo, Rizzi, TCS, 2021]

But we can ask a more complex question now:

Let $\mathrm{bwt}_{\$} : \Sigma^n \to \Sigma^n$, $T \mapsto \mathrm{bwt}(T\$)$ without the dollar.

ex. `banana` ↦ `annbaa`, since bwt(`banana$`) = `annb$aa`.

Questions:

- Is $\mathrm{bwt}_{\$}$ bijective? (no)
- Can we characterize $\mathrm{bwt}_{\$}$-images?
- If W is a $\mathrm{bwt}_{\$}$-image, how many distinct T's map to it?
- How can we find these T's?

When a dollar makes a BWT

Question: Is W a $\text{bwt}_{\$}$-image? In other words, can we insert \$ somewhere to make it a BWT?

When a dollar makes a BWT

Question: Is W a bwt$_\$$-image? In other words, can we insert \$ somewhere to make it a BWT?

Ex.: $W =$ annbaa.

0	\$annbaa	-
1	a\$nnbaa	-
2	an\$nbaa	-
3	ann\$baa	-
4	annb\$aa	bwt(banana\$)
5	annba\$a	-
6	annbaa\$	bwt(nabana\$)

We call 4 and 6 nice positions.

annbaa is a bwt$_\$$-image ✓
with 2 nice positions.

When a dollar makes a BWT

Question: Is W a bwt$_\$$-image? In other words, can we insert \$ somewhere to make it a BWT?

Ex.: $W =$ annbaa.

0	\$annbaa	-
1	a\$nnbaa	-
2	an\$nbaa	-
3	ann\$baa	-
4	annb\$aa	bwt(banana\$)
5	annba\$a	-
6	annbaa\$	bwt(nabana\$)

We call 4 and 6 nice positions.

annbaa is a bwt$_\$$-image ✓
with 2 nice positions.

Ex.: $W =$ banana.

0	\$banana	-
1	b\$anana	-
2	ba\$nana	-
3	ban\$ana	-
4	bana\$na	-
5	banan\$a	-
6	banana\$	-

banana is no bwt$_\$$-image. ✗

Computing nice positions

- Simple algorithm: for every i, $0 \leq i < n$, try reversing: $\mathcal{O}(n^2)$ time
- Our algorithm: $\mathcal{O}(n \log n)$ time
- def.: π_i standard permutation of W with \$ in position i
- idea: compute π_{i+1} directly from π_i in $\mathcal{O}(\log n)$ time
- smart data structure for maintaining permutations

Our algorithm

Lemma: We can get π_{i+1} from π_i with one transposition:

$$\pi_{i+1} \quad = \quad (\pi_i(i), \pi_i(i+1)) \circ \pi_i \underset{\text{\$ is in position } i}{=} (0, \pi_i(i+1)) \circ \pi_i .$$

Our algorithm

Lemma: We can get π_{i+1} from π_i with one transposition:

$$\pi_{i+1} \quad = \quad (\pi_i(i), \pi_i(i+1)) \circ \pi_i \underset{\text{\$ is in position } i}{=} (0, \pi_i(i+1)) \circ \pi_i.$$

Lemma

1. Transposition of elements in **distinct** cycles **merges** the two cycles
2. Transposition of elements in the **same** cycle **splits** the cycle

Our algorithm

1. Transposition of elements in **distinct** cycles **merges** the two cycles

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

Our algorithm

1. Transposition of elements in **distinct** cycles **merges** the two cycles

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

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

Our algorithm

1. Transposition of elements in **distinct** cycles **merges** the two cycles

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

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

2. Transposition of elements in the **same** cycle **splits** the cycle

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

Our algorithm

1. Transposition of elements in **distinct** cycles **merges** the two cycles

   $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 5 & 6 & 4 & 1 & 2 & 3 \end{pmatrix} = (0)(1, 5, 2, 6, 3, 4)$

   $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 0 & 6 & 4 & 1 & 2 & 3 \end{pmatrix} = (0, 5, 2, 6, 3, 4, 1)$

2. Transposition of elements in the **same** cycle **splits** the cycle

   $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 0 & 6 & 4 & 1 & 2 & 3 \end{pmatrix} = (0, 5, 2, 6, 3, 4, 1)$

   $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 6 & 0 & 4 & 1 & 2 & 3 \end{pmatrix} = (0, 5, 2)(6, 3, 4, 1)$

Our algorithm

Ex.: Algorithm **findNicePositions(W)** on $W = \texttt{annbaa}$:

Our algorithm

Ex.: Algorithm **findNicePositions(W)** on $W = \texttt{annbaa}$:

0 $\texttt{\$annbaa}$ $\pi_0 = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 5 & 6 & 4 & 2 & 3 \end{smallmatrix}\right) = (0)(1)(2,5)(3,6)(4)$ merge

Our algorithm

Ex.: Algorithm **findNicePositions(W)** on $W = \texttt{annbaa}$:

0 `$annbaa` $\pi_0 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 5 & 6 & 4 & 2 & 3 \end{pmatrix} = (0)(1)(2,5)(3,6)(4)$ merge

1 `a$nnbaa` $\pi_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 5 & 6 & 4 & 2 & 3 \end{pmatrix} = (0,1)(2,5)(3,6)(4)$ merge

Our algorithm

Ex.: Algorithm **findNicePositions(W)** on $W = \texttt{annbaa}$:

0 `$annbaa` $\pi_0 = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 5 & 6 & 4 & 2 & 3 \end{smallmatrix}\right) = (0)(1)(2,5)(3,6)(4)$ merge

1 `a$nnbaa` $\pi_1 = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 5 & 6 & 4 & 2 & 3 \end{smallmatrix}\right) = (0,1)(2,5)(3,6)(4)$ merge

2 `an$nbaa` $\pi_2 = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 5 & 0 & 6 & 4 & 2 & 3 \end{smallmatrix}\right) = (0,1,5,2)(3,6)(4)$ merge

Our algorithm

Ex.: Algorithm **findNicePositions(W)** on $W = \texttt{annbaa}$:

0 `$annbaa` $\pi_0 = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 5 & 6 & 4 & 2 & 3 \end{smallmatrix}\right) = (0)(1)(2,5)(3,6)(4)$ merge

1 `a$nnbaa` $\pi_1 = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 5 & 6 & 4 & 2 & 3 \end{smallmatrix}\right) = (0,1)(2,5)(3,6)(4)$ merge

2 `an$nbaa` $\pi_2 = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 5 & 0 & 6 & 4 & 2 & 3 \end{smallmatrix}\right) = (0,1,5,2)(3,6)(4)$ merge

3 `ann$baa` $\pi_3 = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 5 & 6 & 0 & 4 & 2 & 3 \end{smallmatrix}\right) = (0,1,5,2,6,3)(4)$ merge

Our algorithm

Ex.: Algorithm **findNicePositions(W)** on $W = \texttt{annbaa}$:

0	`$annbaa`	$\pi_0 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 0&1&5&6&4&2&3 \end{pmatrix} = (0)(1)(2,5)(3,6)(4)$	merge
1	`a$nnbaa`	$\pi_1 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&2&5&6&4&2&3 \end{pmatrix} = (0,1)(2,5)(3,6)(4)$	merge
2	`an$nbaa`	$\pi_2 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&0&6&4&2&3 \end{pmatrix} = (0,1,5,2)(3,6)(4)$	merge
3	`ann$baa`	$\pi_3 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&6&0&4&2&3 \end{pmatrix} = (0,1,5,2,6,3)(4)$	merge
4	annb\$aa	$\pi_4 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&6&4&0&2&3 \end{pmatrix} = (0,1,5,2,6,3,4)$	split

Our algorithm

Ex.: Algorithm **findNicePositions(W)** on $W =$ `annbaa`:

0	`$annbaa`	$\pi_0 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 0&1&5&6&4&2&3 \end{pmatrix} = (0)(1)(2,5)(3,6)(4)$	merge
1	`a$nnbaa`	$\pi_1 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&2&5&6&4&2&3 \end{pmatrix} = (0,1)(2,5)(3,6)(4)$	merge
2	`an$nbaa`	$\pi_2 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&0&6&4&2&3 \end{pmatrix} = (0,1,5,2)(3,6)(4)$	merge
3	`ann$baa`	$\pi_3 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&6&0&4&2&3 \end{pmatrix} = (0,1,5,2,6,3)(4)$	merge
4	`annb$aa`	$\pi_4 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&6&4&0&2&3 \end{pmatrix} = (0,1,5,2,6,3,4)$	split
5	`annba$a`	$\pi_5 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&6&4&2&0&3 \end{pmatrix} = (0,1,5)(2,6,3,4)$	merge

Our algorithm

Ex.: Algorithm **findNicePositions(W)** on $W = \texttt{annbaa}$:

$0 \quad \texttt{\$annbaa} \quad \pi_0 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 0&1&5&6&4&2&3 \end{pmatrix} = (0)(1)(2,5)(3,6)(4)$ merge

$1 \quad \texttt{a\$nnbaa} \quad \pi_1 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&2&5&6&4&2&3 \end{pmatrix} = (0,1)(2,5)(3,6)(4)$ merge

$2 \quad \texttt{an\$nbaa} \quad \pi_2 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&0&6&4&2&3 \end{pmatrix} = (0,1,5,2)(3,6)(4)$ merge

$3 \quad \texttt{ann\$baa} \quad \pi_3 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&6&0&4&2&3 \end{pmatrix} = (0,1,5,2,6,3)(4)$ merge

$\boxed{4 \quad \texttt{annb\$aa} \quad \pi_4 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&6&4&0&2&3 \end{pmatrix} = (0,1,5,2,6,3,4)}$ split

$5 \quad \texttt{annba\$a} \quad \pi_5 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&6&4&2&0&3 \end{pmatrix} = (0,1,5)(2,6,3,4)$ merge

$\boxed{6 \quad \texttt{annbaa\$} \quad \pi_6 = \begin{pmatrix} 0&1&2&3&4&5&6 \\ 1&5&6&4&2&3&0 \end{pmatrix} = (0,1,5,3,4,2,6)}$

Our algorithm

Algorithm **findNicePositions(W)**:

```
$\pi_0 \leftarrow$ standard permutation of $\$W$
$c \leftarrow$ number of cycles of $\pi_0$
$\mathcal{I} \leftarrow \emptyset$
For each position $i$, $0 \leq i < n$:
    if $i+1$ and $i$ in the same cycle then
        $c \leftarrow c+1$                                // split
    otherwise
        $c \leftarrow c-1$                                // merge
    update $\pi_i$ to $\pi_{i+1}$
    if $c = 1$: add $i+1$ to $\mathcal{I}$
return $\mathcal{I}$
```

Our algorithm

Algorithm **findNicePositions(W)**:

```
$\pi_0 \leftarrow$ standard permutation of $\$W$
$c \leftarrow$ number of cycles of $\pi_0$
$\mathcal{I} \leftarrow \emptyset$
For each position $i$, $0 \leq i < n$:
        if $i+1$ and $i$ in the same cycle then
                $c \leftarrow c+1$                                  // split
        otherwise
                $c \leftarrow c-1$                                  // merge
        update $\pi_i$ to $\pi_{i+1}$
        if $c = 1$: add $i+1$ to $\mathcal{I}$
return $\mathcal{I}$
```

Analysis

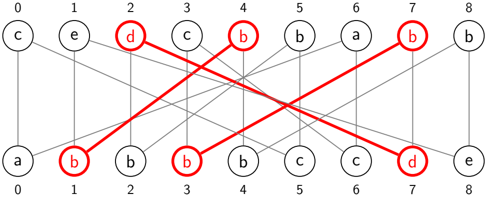
- Using **splay trees** [Sleator and Tarjan, 1985]:
    - decide whether i and $i+1$ in the same cycle in amortized $\mathcal{O}(\log n)$ time
    - update π_i in amortized $\mathcal{O}(\log n)$ time
- Altogether $\mathcal{O}(n \log n)$ time

Characterizing nice positions

Def.
$P = P_{left} \dot{\cup} P_{right}$ is called **pseudo-cycle** if $P_{left} < P_{right}$ and $\pi(P) = (P_{left} - 1) \cup P_{right}$.

ex.: $W =$ `cedcbbabb`, then $\pi = \left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 5 & 8 & 7 & 6 & 1 & 2 & 0 & 3 & 4 \end{smallmatrix}\right)$.

$P = \{2,4,7\}$, $\pi(P) = \{1,3,7\}$, $P_{left} = \{2,4\}$, $P_{right} = \{7\}$

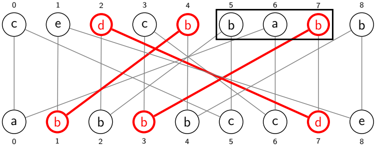


Characterizing nice positions

Why are pseudo-cycles bad?

`cedcbbabb`

$P_{left} = \{2, 4\}$, $P_{right} = \{7\}$



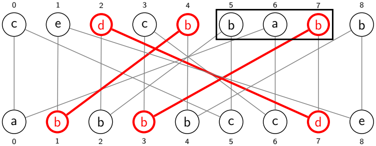
critical interval $= \{5, 6, 7\}$.

Characterizing nice positions

Why are pseudo-cycles bad?

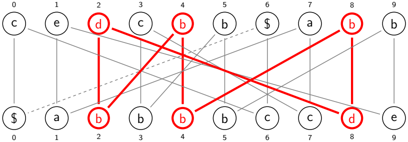
`cedcbbabb`

$P_{left} = \{2, 4\}$, $P_{right} = \{7\}$



critical interval $= \{5, 6, 7\}$.

`cedcbb$abb`



Red edges become cyles in π_6

Characterizing nice positions

Thm. Position i is nice iff there is no pseudo-cycle in π whose critical interval contains i.

4. BWT of string collections

How to compute the BWT of a set of strings?

[Cenzato and L., CPM 2022]

ex. $\mathcal{M} = \{\texttt{ATATG}, \texttt{TGA}, \texttt{ACG}, \texttt{ATCA}, \texttt{GGA}\}$

It turns out that there are many non-equivalent methods in use:

variant (our terminology)	result on example	tools
eBWT	CGGGATGTACGTTAAAAA	pfpebwt
dollarEBWT	GGAAACGG$$$TTACTGT$AAA$	G2BWT, pfpebwt, msbwt
multidolBWT	GAGAAGCG$$$TTATCTG$AAA$	BCR, ropebwt2, nvSetBWT, Merge-BWT, eGSA, eGAP, bwt-lcp-parallel, gsufsort
concatBWT	$AAGAGGGC$#$TTACTGT$AAA$	BigBWT, tools for single strings
colexBWT	AAAGGCGG$$$TTACTGT$AAA$	ropebwt2

The different BWT variants

1. eBWT($\mathcal{M}$): the extended BWT of $\mathcal{M}$ of Mantaci et al. (2007)
   uses omega-order instead of lexicographical order: e.g. $\texttt{aba} <_\omega \texttt{ab}$

The different BWT variants

1. eBWT($\mathcal{M}$): the extended BWT of $\mathcal{M}$ of Mantaci et al. (2007)
   uses omega-order instead of lexicographical order: e.g. aba $<_\omega$ ab
   $T <_\omega S$ if (a) $T^\omega < S^\omega$, or (b) $T^\omega = S^\omega,\ T = U^k, S = U^m$ and $k < m$

The different BWT variants

1. eBWT($\mathcal{M}$): the extended BWT of $\mathcal{M}$ of Mantaci et al. (2007)
   uses omega-order instead of lexicographical order: e.g. `aba` $<_\omega$ `ab`
   $T <_\omega S$ if (a) $T^\omega < S^\omega$, or (b) $T^\omega = S^\omega$, $T = U^k, S = U^m$ and $k < m$
2. dollarEBWT($\mathcal{M}$) $=$ eBWT($\{ T_i\$ \ : \ T_i \in \mathcal{M}\}$)

The different BWT variants

1. eBWT($\mathcal{M}$): the extended BWT of $\mathcal{M}$ of Mantaci et al. (2007)
   uses omega-order instead of lexicographical order: e.g. aba $<_\omega$ ab
   $T <_\omega S$ if (a) $T^\omega < S^\omega$, or (b) $T^\omega = S^\omega$, $T = U^k, S = U^m$ and $k < m$
2. dollarEBWT($\mathcal{M}$) $=$ eBWT($\{ T_i\$ \ : \ T_i \in \mathcal{M}\}$)
3. multidolBWT($\mathcal{M}$) $=$ bwt($T_1\$_1 T_2\$_2 \cdots T_k\$_k$), where dollars are smaller than characters from Σ, and $\$_1 < \$_2 < \ldots < \$_k$

The different BWT variants

1. eBWT($\mathcal{M}$): the extended BWT of $\mathcal{M}$ of Mantaci et al. (2007)
   uses omega-order instead of lexicographical order: e.g. aba $<_\omega$ ab
   $T <_\omega S$ if (a) $T^\omega < S^\omega$, or (b) $T^\omega = S^\omega$, $T = U^k, S = U^m$ and $k < m$
2. dollarEBWT($\mathcal{M}$) $=$ eBWT($\{T_i\$ \ : \ T_i \in \mathcal{M}\}$)
3. multidolBWT($\mathcal{M}$) $=$ bwt($T_1\$_1 T_2\$_2 \cdots T_k\$_k$), where dollars are smaller than characters from Σ, and $\$_1 < \$_2 < \ldots < \$_k$
4. concatBWT($\mathcal{M}$) $=$ bwt($T_1\$ T_2\$ \cdots T_k\$\#$), where $\# < \$$

The different BWT variants

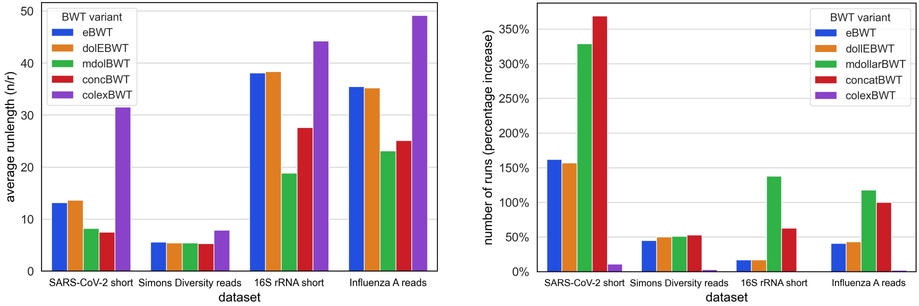
1. eBWT($\mathcal{M}$): the extended BWT of $\mathcal{M}$ of Mantaci et al. (2007) uses omega-order instead of lexicographical order: e.g. $\texttt{aba} <_\omega \texttt{ab}$
   $T <_\omega S$ if (a) $T^\omega < S^\omega$, or (b) $T^\omega = S^\omega$, $T = U^k, S = U^m$ and $k < m$
2. dollarEBWT($\mathcal{M}$) $=$ eBWT($\{T_i\$ \,:\, T_i \in \mathcal{M}\}$)
3. multidolBWT($\mathcal{M}$) $= \text{bwt}(T_1\$_1 T_2\$_2 \cdots T_k\$_k)$, where dollars are smaller than characters from Σ, and $\$_1 < \$_2 < \ldots < \$_k$
4. concatBWT($\mathcal{M}$) $= \text{bwt}(T_1\$T_2\$ \cdots T_k\$\#)$, where $\# < \$$
5. colexBWT($\mathcal{M}$) $= \text{multidol}(\mathcal{M}, \gamma)$, where γ is the permutation corresponding to the colexicographic ('reverse lexicographic').

The different BWT variants

BWT variant	example	order of shared suffixes
non-sep.based		
eBWT($\mathcal{M}$)	CGGGATGTACGTTAAAAA	omega-order of strings
separator-based		
dollarEBWT($\mathcal{M}$)	GGAAACGG$$$TTACTGT$AAA$	lexicographic order of strings
multidolBWT($\mathcal{M}$)	GAGAAGCG$$$TTATCTG$AAA$	input order of strings
concatBWT($\mathcal{M}$)	AAGAGGGC$$$TTACTGT$AAA$	lexicographic order of subsequent strings in input
colexBWT($\mathcal{M}$)	AAAGGCGG$$$TTACTGT$AAA$	colexicographic order

The different BWT variants

Results regarding r on short sequence datasets, of all BWT variants.



Left: average runlength (n/r). Right: number of runs r (percentage increase with respect to the optimal BWT of [Bentley et al., ESA 2020]).

The different BWT variants

- BWT variants differ significantly among each other ($>$ 11% Hamming distance on some data sets)
- we theoretically explained these differences ("interesting intervals")
- differences especially high on large sets of short sequences
- multidolBWT and concatBWT depend on the input order
- differences extend to parameter r (number of runs of the BWT) (up to a factor of 4.2 in our experiments)

Part III:

Conclusion

Dollar or no dollar, that is the question.

Conclusion

The two definitions of the BWT (with and without dollar) are non-equivalent. In particular,

Conclusion

The two definitions of the BWT (with and without dollar) are non-equivalent. In particular,

1. differences in the transform itself: $r(T)$ vs. $r(T\$)$

Conclusion

The two definitions of the BWT (with and without dollar) are non-equivalent. In particular,

1. differences in the transform itself: $r(T)$ vs. $r(T\$)$
2. BWT construction: cannot use SA when no dollar

Conclusion

The two definitions of the BWT (with and without dollar) are non-equivalent. In particular,

1. differences in the transform itself: $r(T)$ vs. $r(T\$)$
2. BWT construction: cannot use SA when no dollar
3. BWT images: $\mathrm{bwt}_\$$ vs. bwt

Conclusion

The two definitions of the BWT (with and without dollar) are non-equivalent. In particular,

1. differences in the transform itself: $r(T)$ vs. $r(T\$)$
2. BWT construction: cannot use SA when no dollar
3. BWT images: $\mathrm{bwt}_{\$}$ vs. bwt
4. BWT of string collections: several non-equivalent methods in use

Some open problems

- Is the factor between $r(T)$ and $r(T\$)$ additive or multiplicative?

Some open problems

- Is the factor between $r(T)$ and $r(T\$)$ additive or multiplicative?
- Characterize $\mathrm{bwt}_{\$}$-images (for bwt: Thm. of Likhomanov & Shur)

Some open problems

- Is the factor between $r(T)$ and $r(T\$)$ additive or multiplicative?
- Characterize $\text{bwt}_\$$-images (for bwt: Thm. of Likhomanov & Shur)
- Find combinatorial characterization of strings with same $\text{bwt}_\$$ (for bwt: conjugates) e.g. $\text{bwt}_\$(\texttt{abbba}) = \text{bwt}_\$(\texttt{babba}) = \texttt{abbba}$

Some open problems

- Is the factor between $r(T)$ and $r(T\$)$ additive or multiplicative?
- Characterize $\text{bwt}_\$$-images (for bwt: Thm. of Likhomanov & Shur)
- Find combinatorial characterization of strings with same $\text{bwt}_\$$ (for bwt: conjugates) e.g. $\text{bwt}_\$(\texttt{abbba}) = \text{bwt}_\$(\texttt{babba}) = \texttt{abbba}$
- Use pseudo-cycles for computing nice positions (first steps in [Giuliani, L., Masillo, ICTCS 2022])

Acknowledgements (co-authors of the work presented)

Marinella Sciortino
(Univ. of Palermo)

Romeo Rizzi
(Univ. of Verona)

Shunsuke Inenanaga
(Kyushu Univ.)

Christina Boucher
(Univ. of Florida)

Massimiliano Rossi
(Illumina Inc.)

Sara Giuliani
(Univ. of Verona)

Davide Cenzato
(Univ. of Verona)

Francesco Masillo
(Univ. of Verona)

Literature

- S. Giuliani, Zs. Lipták, F. Masillo, R. Rizzi: When a dollar makes a BWT. Theor. Comput. Sci. 857: 123-146 (2021).
- S. Giuliani, Zs. Lipták, F. Masillo: When a Dollar in a Fully Clustered Word Makes a BWT, ICTCS 2022.
- S. Giuliani, S. Inenaga, Zs. Lipták, M. Sciortino: On bit catastrophes for the Burrows-Wheeler-Transform, forthcoming.
- C. Boucher, D. Cenzato, Zs. Lipták, M. Rossi, M. Sciortino, Computing the original eBWT faster, simpler, and with less memory. SPIRE 2021.
- D. Cenzato and Zs. Lipták: A theoretical and experimental analysis of BWT variants for string collections, CPM 2022.
- D. Cenzato and Zs. Lipták: Computing the optimal BWT using SAIS, WCTA 2022.

Thank you for your attention!

email: **zsuzsanna.liptak@univr.it**