On the Burrows-Wheeler Transform of string collections

Zsuzsanna Lipták

University of Verona (Italy)

Primavera dell'Informatica Teorica
11 Jan. 2024

The Burrows-Wheeler Transform (BWT)

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

The Burrows-Wheeler Transform (BWT)

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

all rotations (conjugates)		all rotations, sorted
banana	$\longrightarrow$ lexicographic order	abanan
ananab		anaban
nanaba		ananab
anaban		banana
nabana		nabana
abanan		nanaba

The Burrows-Wheeler Transform (BWT)

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

all rotations (conjugates)		all rotations, sorted
banana	$\longrightarrow$ lexicographic order	abanan
ananab		anaban
nanaba		ananab
anaban		banana
nabana		nabana
abanan		nanaba

A (non-efficient) algorithm: List all of rotations of string T, sort them lexicographically, concatenate last characters: bwt(banana) = nnbaaa

AWARDS & RECOGNITION

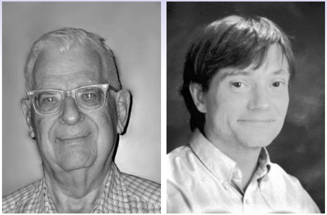
Inventors of BW-transform and the FM-index Receive Kanellakis Award

2022

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.

source: `https://awards.acm.org/kanellakis`

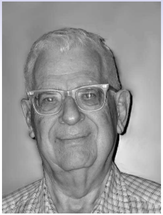
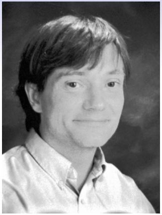
The BWT



source: Adjeroh, Bell, Mukerjee (2008)

- introduced by M. Burrows and D. Wheeler in 1994 as a lossless text compression algorithm
- P. Ferragina and G. Manzini showed later how to use it for pattern matching, leading to the FM-index [FOCS, 2000; JACM 2005]
- recent: *r*-index [Gagie et al, JACM 2020; Bannai et al. TCS 2020]

The BWT

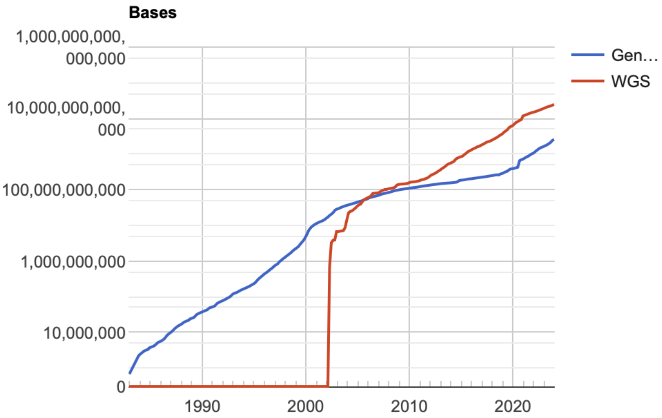
source: Adjeroh, Bell, Mukerjee (2008)

- introduced by M. Burrows and D. Wheeler in 1994 as a lossless text compression algorithm
- P. Ferragina and G. Manzini showed later how to use it for pattern matching, leading to the FM-index [FOCS, 2000; JACM 2005]
- recent: r-index [Gagie et al, JACM 2020; Bannai et al. TCS 2020]

Some properties of the BWT:

- computable in linear time $\mathcal{O}(n)$ $\qquad n = |T|$
- reversible in linear time $\mathcal{O}(n)$
- uncompressed: same space as text
- if T repetitive, good for compression (see later)

GenBank and WGS Statistics



From strings to string collections

- Human Genome Project (first draft: 2000, completion: 2003)

From strings to string collections

- Human Genome Project (first draft: 2000, completion: 2003)
- Studying variation:
  - 1,000 Genomes Project (human): 2008-2015
  - 1001 Genomes (Arabidopsis thaliana)
  - 3,000 Rice Genomes Project
  - 100,000 Genomes Project (human, completed 2018)

From strings to string collections

- Human Genome Project (first draft: 2000, completion: 2003)
- Studying variation:
  - 1,000 Genomes Project (human): 2008-2015
  - 1001 Genomes (Arabidopsis thaliana)
  - 3,000 Rice Genomes Project
  - 100,000 Genomes Project (human, completed 2018)
- Population-wide:
  - Faroe Genome Project: sequence all 50,000 people
  - Sequencing Iceland (325,000 people): $>$ 57,000 sequenced

From strings to string collections

- Human Genome Project (first draft: 2000, completion: 2003)
- Studying variation:
  - 1,000 Genomes Project (human): 2008-2015
  - 1001 Genomes (Arabidopsis thaliana)
  - 3,000 Rice Genomes Project
  - 100,000 Genomes Project (human, completed 2018)
- Population-wide:
  - Faroe Genome Project: sequence all 50,000 people
  - Sequencing Iceland (325,000 people): $>$ 57,000 sequenced
- Human diversity:
  - Genes & Health in East London: 100,000 people of Bangladeshi and Pakistani origin
  - Sequencing African genomes (Nature 2020)
  - Sequencing indigenous Australian genomes (Nature 2023)

From strings to string collections

- Human Genome Project (first draft: 2000, completion: 2003)
- Studying variation:
  - 1,000 Genomes Project (human): 2008-2015
  - 1001 Genomes (Arabidopsis thaliana)
  - 3,000 Rice Genomes Project
  - 100,000 Genomes Project (human, completed 2018)
- Population-wide:
  - Faroe Genome Project: sequence all 50,000 people
  - Sequencing Iceland (325,000 people): $>$ 57,000 sequenced
- Human diversity:
  - Genes & Health in East London: 100,000 people of Bangladeshi and Pakistani origin
  - Sequencing African genomes (Nature 2020)
  - Sequencing indigenous Australian genomes (Nature 2023)
- SARS-CoV-2 viral sequences

From strings to string collections

Our data is

- growing rapidly, and
- changing: **from individual strings to string collections**
- many of these consist of **many similar copies** of the same string

Outline of talk

- The Burrows-Wheeler Transform (BWT)
- The extended BWT (eBWT)
- Other variants of the BWT for string collections
- Why does it matter?
- Conclusions

The Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT)

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

all rotations (conjugates)		all rotations, sorted
`banana`		`abanan`
`ananab`	$\longrightarrow$	`anaban`
`nanaba`	lexicographic order	`ananab`
`anaban`		`banana`
`nabana`		`nabana`
`abanan`		`nanaba`

Why is the BWT useful in text compression?

BWT-matrix (F = first column, L = last column)

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

Why is the BWT useful in text compression?

BWT-matrix (F = first column, L = last column)

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

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

Why is the BWT useful in text compression?

BWT-matrix (F = first column, L = last column)

	F					L
1	a	b	a	n	a	n
2	a	n	a	b	a	n
3	a	n	a	n	a	b
4	b	a	n	a	n	a
5	n	a	b	a	n	a
6	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 (cyclically): $T = \underset{1\,2\,3\,4\,5\,6}{\texttt{banana}}$

Why is the BWT useful in text compression?

BWT-matrix (F = first column, L = last column)

	F					L
1	a	b	a	n	a	n
2	a	n	a	b	a	n
3	a	n	a	n	a	b
4	b	a	n	a	n	a
5	n	a	b	a	n	a
6	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 (cyclically):
  $T = \underset{1}{\mathtt{b}}\underset{2}{\mathtt{a}}\underset{3}{\mathtt{n}}\underset{4}{\mathtt{a}}\underset{5}{\mathtt{n}}\underset{6}{\mathtt{a}}$
- **Obs. 3:** all occurrences of a substring appear in consecutive rows as prefix

Why is the BWT useful in text compression?

- **Obs. 1:** F = characters of T in lexicographic order
- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows as prefix

Why is the BWT useful in text compression?

- **Obs. 1:** F = characters of T in lexicographic order
- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows as prefix

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

2 occ's of `ana`

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

Why is the BWT useful in text compression?

- **Obs. 1:** F = characters of T in lexicographic order
- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows as prefix

Ex.: T = banana has 2 occurrences of the substring 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 is the BWT useful in text compression?

- **Obs. 1:** F = characters of T in lexicographic order
- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows as prefix

Ex.: T = banana has 2 occurrences of the substring 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 is the BWT useful in text compression?

- **Obs. 1:** F = characters of T in lexicographic order
- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows as prefix

Ex.: T = banana has 2 occurrences of the substring 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 is the BWT useful in text compression?

- **Obs. 1:** F = characters of T in lexicographic order
- **Obs. 2:** L_i precedes F_i in T
- **Obs. 3:** all occurrences of a substring appear in consecutive rows as prefix

Ex.: T = banana has 2 occurrences of the substring 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 in general:

Of course, things are a bit more complicated in general:

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)

Of course, things are a bit more complicated in general:

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)

Compression with the BWT

- takes advantage of this 'clustering effect'

Compression with the BWT

- takes advantage of this 'clustering effect'
- **Def.:** $r(T)$ = number of runs of $\mathrm{bwt}(T)$ (run: maximal equal-letter run)

Ex.: $r(\texttt{banana}) = 3$
$\mathrm{bwt}(\texttt{banana}) = \texttt{nnbaaa}$

Compression with the BWT

- takes advantage of this 'clustering effect'
- **Def.:** $r(T)$ = number of runs of bwt(T) (run: maximal equal-letter run)

  Ex.: $r(\texttt{banana}) = 3$
  bwt(`banana`) = `nnbaaa`
- **compression with BWT:**
  uses runlength-encoding (RLE)

Compression with the BWT

- takes advantage of this 'clustering effect'
- **Def.:** $r(T)$ = number of runs of bwt(T) (run: maximal equal-letter run)

  Ex.: $r(\texttt{banana}) = 3$
  bwt(`banana`) = `nnbaaa`

- **compression with BWT:**
  uses runlength-encoding (RLE)

  replace each run by (char,int)-pair
  RLE(`bbbbbbbbcaaaaaaaaaaabb`) = `b8c1a11b2`

Compression with the BWT

- takes advantage of this 'clustering effect'
- **Def.:** $r(T)$ = number of runs of bwt(T) (run: maximal equal-letter run)

  Ex.: $r(\texttt{banana}) = 3$
  bwt(`banana`) = `nnbaaa`
- **compression with BWT:**
  uses runlength-encoding (RLE)

  replace each run by (char,int)-pair
  RLE(`bbbbbbbbcaaaaaaaaaaabb`) = `b8c1a11b2`

  Compression: $T \mapsto \underbrace{\text{RLE(bwt}(T))}_{\text{storage space: } O(r)}$

  Ex.: `banana` $\mapsto$ `n2b1a3`

Compression with the BWT

- takes advantage of this 'clustering effect'
- **Def.:** $r(T)$ = number of runs of $\mathrm{bwt}(T)$ (run: maximal equal-letter run)

  Ex.: $r(\texttt{banana}) = 3$
  $\mathrm{bwt}(\texttt{banana}) = \texttt{nnbaaa}$

- **compression with BWT:**
  uses runlength-encoding (RLE)

  replace each run by (char,int)-pair
  $\mathrm{RLE}(\texttt{bbbbbbbbcaaaaaaaaaaabb}) = \texttt{b8c1a11b2}$

  Compression: $T \mapsto \underbrace{\mathrm{RLE}(\mathrm{bwt}(T))}_{\text{storage space: } O(r)}$

  Ex.: $\texttt{banana} \mapsto \texttt{n2b1a3}$

- good if r is much smaller than $n = |T|$
  (i.e. if few runs)

Compression with the BWT

- takes advantage of this 'clustering effect'
- **Def.:** $r(T)$ = number of runs of $\mathrm{bwt}(T)$ (run: maximal equal-letter run)

Ex.: $r(\texttt{banana}) = 3$
$\mathrm{bwt}(\texttt{banana}) = \texttt{nnbaaa}$

- **compression with BWT:**
uses runlength-encoding (RLE)

  replace each run by (char,int)-pair
  $\mathrm{RLE}(\texttt{bbbbbbbbcaaaaaaaaaaabb}) = \texttt{b8c1a11b2}$

  Compression: $T \mapsto \underbrace{\mathrm{RLE}(\mathrm{bwt}(T))}_{\text{storage space: } O(r)}$

  Ex.: $\texttt{banana} \mapsto \texttt{n2b1a3}$

- good if r is much smaller than $n = |T|$
(i.e. if few runs)
- for **repetitive strings**, r is small
(repetitive: many repeated substrings)

Reversing the BWT (lossless compression)

input: nnbaaa, 4 — bwt(T), i: where $1 \leq i \leq n$

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

Reversing the BWT (lossless compression)

input: nnbaaa, 4 — bwt(T), i: where $1 \leq i \leq n$

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

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

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

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

Reversing the BWT (lossless compression)

input: `nnbaaa`, 4 — bwt(T), i: where $1 \leq i \leq n$

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

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

	F	L
1	abana	n
2	anaba	n
3	anana	b
4	banan	a
5	naban	a
6	nanab	a

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

Recall:

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

Obs. 2: for all i: L_i precedes F_i in T.

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`, 4

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, 4

	L
1	n
2	n
3	b
4	a
5	a
6	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`, 4

	F	L
1	a	n
2	a	n
3	a	b
4	b	a
5	n	a
6	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, 4

	F	L	
1	a	n	
2	a	n	
3	a	b	a
4	b	a	
5	n	a	
6	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`, 4

	F	L
1	a	n
2	a	n
3	a	b
4	b	a
5	n	a
6	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, 4

	F	L
1	a	n
2	a	n
3	a	b
4	b	a
5	n	a
6	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`, 4

	F	L
1	a	n
2	a	n
3	a	b
4	b	a
5	n	a
6	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, 4

	F	L
1	a	n
2	a	n
3	a	b
4	b	a
5	n	a
6	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`, 4

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

b a n a n a

The BWT of string collections

- The BWT is good on repetitive strings.
- Our string collections are highly repetitive:
  many similar copies of the same string
- But: how do we compute the BWT of a multiset?

The BWT of string collections

- The BWT is good on repetitive strings.
- Our string collections are highly repetitive:
  many similar copies of the same string
- But: how do we compute the BWT of a multiset?

Generalization of the BWT to multisets:
the extended BWT (eBWT) (next)

The extended BWT

The extended BWT

[Mantaci, Restivo, Rosone, Sciortino, TCS, 2007]

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

all rotations (conjugates)		all rotations, sorted	
bana		aban	n
anab	$\longrightarrow$	anab	b
naba	omega order	an	n
aban		bana	a
an		naba	a
na		na	a

N.B. anab $<_\omega$ 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

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 $k < m$

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

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

- omega-order instead of lex-order
- same as lex-order if neither string is prefix of the other
- omega-order necessary for the LF-property
- the eBWT inherits BWT properties: clustering effect, reversibility, useful for lossless text compression, efficient pattern matching, ...
- However, until recently no linear-time algorithm known.

The extended BWT

- omega-order instead of lex-order
- same as lex-order if neither string is prefix of the other
- omega-order necessary for the LF-property
- the eBWT inherits BWT properties: clustering effect, reversibility, useful for lossless text compression, efficient pattern matching, ...
- However, until recently no linear-time algorithm known.

2021:

- linear-time algorithm [Bannai, Kärkkäinen, Köppl, Piatkowski, CPM 2021]
- We simplified this algorithm, and
- gave first efficient implementations of the eBWT: tools `pfpebwt,cais`

  [Boucher, Cenzato, L., Rossi, Sciortino, SPIRE 2021]

Other BWT variants for string collections

The BWT of string collections

[Cenzato and L., CPM 2022, Arxiv 2023]

Question: How do dedicated tools compute the BWT of a string collection? (string collection: multiset of strings)

- We studied 18 publicly available tools.
- Only ours compute the eBWT (`pfpebwt,cais`).
- We identified 4 more non-equivalent approaches: the resulting BWTs are all different.
- Often the method is not explicitly stated.
- **Underlying assumption: they are all the same.**
- But they differ a lot (Hamming distance, number of runs).
- **N.B.:** all BWT variants are correct (LF-property, . . .)

The other BWT variants for string collections

The different approaches are:

1. extended BWT of strings with terminator symbol \$ (dollarEBWT)
2. concatenate strings, separating them with different dollars (multidolBWT)
3. first sort colexicographically, then do 2. (colexBWT)
4. concatenate strings, separating them with same dollar (concatBWT)

All use terminator / separator symbols ('dollars'). So we call them separator-based BWT variants.

The BWT variants for string collections

Ex. $\mathcal{M} = \{\mathtt{ATATG}, \mathtt{TGA}, \mathtt{ACG}, \mathtt{ATCA}, \mathtt{GGA}\}$

variant (our terminology)	result on example	tools
eBWT	CGGGATGTACGTTAAAAA	pfpebwt,cais
dollarEBWT	GGAAACGG$$$TTACTGTAAA	G2BWT, msbwt
multidolBWT	GAGAAGCG$$$TTATCTGAAA	gsufsort, ropebwt2, eGSA, Merge-BWT, eGAP, nvSetBWT, BCR-LCP-GSA, grlBWT, BEETL, bwt-lcp-parallel
colexBWT	AAAGGCGG$$$TTACTGTAAA	ropebwt2, BCR-LCP-GSA
concatBWT	$AAGAGGGC$#$TTACTGT$AAA$	BigBWT, r-pfbwt, CMS-BWT tools for single strings

The dollar-eBWT

1. dollarEBWT($\mathcal{M}$) = eBWT($\{T_i\$: T_i \in \mathcal{M}\}$), $\quad \$ < c$ for all char's c

Now no string is prefix of another $\implies$ omega-order same as lex-order.

$\mathcal{M} = \{\mathtt{bana\$}, \mathtt{an\$}\}$

dollarEBWT

\$an	n
\$bana	a
a\$ban	n
an\$	\$
ana\$b	b
bana\$	\$
n\$a	a
na\$ba	a

nan\$b\$aa

The dollar-eBWT

1. dollarEBWT($\mathcal{M}$) = eBWT($\{T_i\$: T_i \in \mathcal{M}\}$), $\$ < c$ for all char's c

Now no string is prefix of another $\implies$ omega-order same as lex-order.

$\mathcal{M} = \{\mathtt{bana\$}, \mathtt{an\$}\}$

dollarEBWT

\$an	n
\$bana	a
a\$ban	n
an\$	\$
ana\$b	b
bana\$	\$
n\$a	a
na\$ba	a

nan\$b\$aa

eBWT of $\{\mathtt{bana}, \mathtt{an}\}$

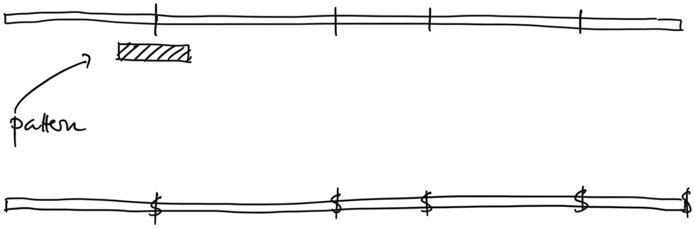
aban	n
anab	b
an	n
bana	a
naba	a
na	a

nbnaaa

The different BWT variants

The other 3 methods concatenate the input strings, and then apply the classical BWT.

The main issue here is to avoid spurious substrings:



The multidollar BWT

2. multidolBWT($\mathcal{M}$) $=$ bwt($T_1\$_1 T_2\$_2 \cdots T_k\$_k$), where dollars are smaller than characters from Σ, and $\$_1 < \$_2 < \ldots < \$_k$

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

bwt($\texttt{ATATG}\$_1\texttt{TGA}\$_2\texttt{ACG}\$_3\texttt{ATCA}\$_4\texttt{GGA}\$_5$) $=$ GAGAAGCG\$\$TTATCTG\$AAA\$

The multidollar BWT

2. multidolBWT($\mathcal{M}$) = bwt($T_1\$_1 T_2\$_2 \cdots T_k\$_k$), where dollars are smaller than characters from Σ, and $\$_1 < \$_2 < \ldots < \$_k$

Ex. $\mathcal{M} = \{\mathtt{ATATG}, \mathtt{TGA}, \mathtt{ACG}, \mathtt{ATCA}, \mathtt{GGA}\} \rightsquigarrow$

bwt($\mathtt{ATATG\$_1TGA\$_2ACG\$_3ATCA\$_4GGA\$_5}$) = $\mathtt{GAGAAGCG\$\$\$TTATCTG\$AAA\$}$

- most commonly used method
- analogous to Generalized Suffix Tree and Generalized Suffix Array
- dollars are different only conceptually (break ties by index)
- equivalent: concatenate without separators, use bitstring marking string beginnings

The colex BWT

3. colexBWT($\mathcal{M}$): multidolBWT of the strings in colexicographic order

colex order $=$ lexicographic order of the reverse strings

The colex BWT

3. colexBWT($\mathcal{M}$): multidolBWT of the strings in colexicographic order

colex order = lexicographic order of the reverse strings

Ex. $\mathcal{M} = \{\mathtt{ATATG}, \mathtt{TGA}, \mathtt{ACG}, \mathtt{ATCA}, \mathtt{GGA}\}$

colex order: ATCA, GGA, TGA, ACG, ATATG $\rightsquigarrow$

bwt($\mathtt{ATCA\$_1GGA\$_2TGA\$_3ACG\$_4ATATG\$_5}$) $=$ AAAGGCGG$$TTACTGTAAA

The colex BWT

3. colexBWT($\mathcal{M}$): multidolBWT of the strings in colexicographic order

colex order = lexicographic order of the reverse strings

Ex. $\mathcal{M} = \{\mathtt{ATATG}, \mathtt{TGA}, \mathtt{ACG}, \mathtt{ATCA}, \mathtt{GGA}\}$

colex order: ATCA,GGA,TGA,ACG,ATATG $\rightsquigarrow$

bwt($\mathtt{ATCA\$_1GGA\$_2TGA\$_3ACG\$_4ATATG\$_5}$) = AAAGGCGG$$TTACTGTAAA

- reduces number of runs (see later)
- implemented as an option in ropebwt2,BCR-LCP-GSA

The concat BWT

4. concatBWT$(\mathcal{M}) = \text{bwt}(T_1\$T_2\$\cdots T_k\$\#)$, where $\# < \$$

The concat BWT

4. concatBWT$(\mathcal{M}) = \text{bwt}(T_1\$T_2\$\cdots T_k\$\#)$, where $\# < \$$

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

$\text{bwt}(\texttt{ATATG\$TGA\$ACG\$ATCA\$GGA\$\#}) = \texttt{\$AAGAGGGC\$\#\$TTACTGT\$AAA\$}$

The concat BWT

4. concatBWT($\mathcal{M}$) = bwt($T_1\$T_2\$\cdots T_k\$\#$), where $\# < \$$

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

bwt($\texttt{ATATG\$TGA\$ACG\$ATCA\$GGA\$\#}$) = $\texttt{\$AAGAGGGC\$\#\$TTACTGT\$AAA\$}$

(for easier comparison, we simplify to $\texttt{AAGAGGGC\$\$\$TTACTGT\$AAA\$}$)

The concat BWT

4. concatBWT$(\mathcal{M}) = \text{bwt}(T_1\$T_2\$\cdots T_k\$\#)$, where $\# < \$$

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

$\text{bwt}(\texttt{ATATG\$TGA\$ACG\$ATCA\$GGA\$\#}) = \texttt{\$AAGAGGGC\$\#\$TTACTGT\$AAA\$}$

(for easier comparison, we simplify to `AAGAGGGC$$$TTACTGT$AAA$`)

- very easy to implement
- used e.g. in `BigBWT`,`CMS-BWT`.

Interesting intervals

Q. Where exactly do these BWT variants differ? **A.** in interesting intervals

Interesting intervals

Q. Where exactly do these BWT variants differ? **A.** in interesting intervals

Ex. $\mathcal{M} = \{\mathtt{ATATG}, \mathtt{TGA}, \mathtt{ACG}, \mathtt{ATCA}, \mathtt{GGA}\}$

BWT variant	example
non-sep.based	
eBWT($\mathcal{M}$)	CGGGATGTACGTTAAAAA
separator-based	
dollarEBWT($\mathcal{M}$)	GGAAACGG$$$TTACTGTAAA
multidolBWT($\mathcal{M}$)	GAGAAGCG$$$TTATCTGAAA
colexBWT($\mathcal{M}$)	AAAGGCGG$$$TTACTGTAAA
concatBWT($\mathcal{M}$)	AAGAGGGC$$$TTACTGTAAA

in color: **interesting intervals**

Interesting intervals

Lemma: If two separator-based BWTs differ in position i then $i \in [b, e]$ for some interesting interval $[b, e]$.

Interesting intervals

Lemma: If two separator-based BWTs differ in position i then $i \in [b, e]$ for some interesting interval $[b, e]$.

Def. U is called **a left-maximal shared suffix** if there exist two strings $S_1, S_2 \in \mathcal{M}$ such that U is a suffix of $S_1\$$ and $S_2\$$ and is preceded by different characters in S_1 and S_2. An interval $[b, e]$ is **interesting** if it corresponds to all occurrences of some left-maximal shared suffix U (i.e., its SA-interval).

Ex. $\mathcal{M} = \{\mathtt{ATATG}, \mathtt{TG\underline{A}}, \mathtt{ACG}, \mathtt{ATC\underline{A}}, \mathtt{GG\underline{A}}\}$, $U = \mathtt{A\$}$.

Interesting intervals

Lemma: If two separator-based BWTs differ in position i then $i \in [b, e]$ for some interesting interval $[b, e]$.

Def. U is called **a left-maximal shared suffix** if there exist two strings $S_1, S_2 \in \mathcal{M}$ such that U is a suffix of $S_1\$$ and $S_2\$$ and is preceded by different characters in S_1 and S_2. An interval $[b, e]$ is **interesting** if it corresponds to all occurrences of some left-maximal shared suffix U (i.e., its SA-interval).

Ex. $\mathcal{M} = \{\mathtt{ATATG}, \mathtt{TG\underline{A}}, \mathtt{ACG}, \mathtt{ATC\underline{A}}, \mathtt{GG\underline{A}}\}$, $U = \mathtt{A\$}$.

```
A$ATC   C
A$GG    G
A$TG    G
```

dollarEBWT

Interesting intervals

Lemma: If two separator-based BWTs differ in position i then $i \in [b, e]$ for some interesting interval $[b, e]$.

Def. U is called **a left-maximal shared suffix** if there exist two strings $S_1, S_2 \in \mathcal{M}$ such that U is a suffix of $S_1\$$ and $S_2\$$ and is preceded by different characters in S_1 and S_2. An interval $[b, e]$ is **interesting** if it corresponds to all occurrences of some left-maximal shared suffix U (i.e., its SA-interval).

Ex. $\mathcal{M} = \{\mathtt{ATATG}, \mathtt{TG\underline{A}}, \mathtt{ACG}, \mathtt{ATC\underline{A}}, \mathtt{GG\underline{A}}\}$, $U = \mathtt{A\$}$.

A\$ATC	C	$\mathtt{A\$_2}\cdots$	G
A\$GG	G	$\mathtt{A\$_4}\cdots$	C
A\$TG	G	$\mathtt{A\$_5}\cdots$	G
dollarEBWT		multidolBWT	

Interesting intervals

Lemma: If two separator-based BWTs differ in position i then $i \in [b, e]$ for some interesting interval $[b, e]$.

Def. U is called **a left-maximal shared suffix** if there exist two strings $S_1, S_2 \in \mathcal{M}$ such that U is a suffix of $S_1\$$ and $S_2\$$ and is preceded by different characters in S_1 and S_2. An interval $[b, e]$ is **interesting** if it corresponds to all occurrences of some left-maximal shared suffix U (i.e., its SA-interval).

Ex. $\mathcal{M} = \{\mathtt{ATATG}, \mathtt{TG\underline{A}}, \mathtt{ACG}, \mathtt{ATC\underline{A}}, \mathtt{GG\underline{A}}\}$, $U = \mathtt{A\$}$.

A\$ATC	C	$\mathtt{A\$_2}\cdots$	G	$\mathtt{A\$_1}\cdots$	C
A\$GG	G	$\mathtt{A\$_4}\cdots$	C	$\mathtt{A\$_2}\cdots$	G
A\$TG	G	$\mathtt{A\$_5}\cdots$	G	$\mathtt{A\$_3}\cdots$	G
dollarEBWT		multidolBWT		colexBWT	

Interesting intervals

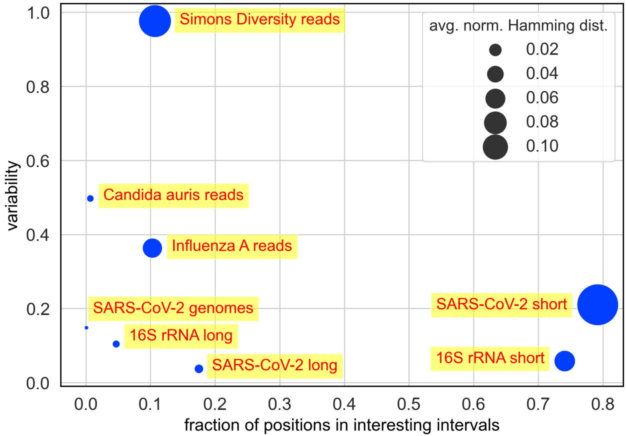
Lemma: If two separator-based BWTs differ in position i then $i \in [b, e]$ for some interesting interval $[b, e]$.

Def. U is called **a left-maximal shared suffix** if there exist two strings $S_1, S_2 \in \mathcal{M}$ such that U is a suffix of $S_1\$$ and $S_2\$$ and is preceded by different characters in S_1 and S_2. An interval $[b, e]$ is **interesting** if it corresponds to all occurrences of some left-maximal shared suffix U (i.e., its SA-interval).

Ex. $\mathcal{M} = \{\mathtt{ATATG}, \mathtt{TG\underline{A}}, \mathtt{ACG}, \mathtt{ATC\underline{A}}, \mathtt{GG\underline{A}}\}$, $U = \mathtt{A\$}$.

A\$ATC	C	$\mathtt{A\$}_2 \cdots$	G	$\mathtt{A\$}_1 \cdots$	C	A\$#	G
A\$GG	G	$\mathtt{A\$}_4 \cdots$	C	$\mathtt{A\$}_2 \cdots$	G	A\$A $\cdots$	G
A\$TG	G	$\mathtt{A\$}_5 \cdots$	G	$\mathtt{A\$}_3 \cdots$	G	A\$G $\cdots$	C
dollarEBWT		multidolBWT		colexBWT		concatBWT	

Hamming distance between separator-based BWTs



Variability

$$var(\mathcal{M}) = \frac{\sum_{[b,e] \text{ interesting int.}} var([b,e])}{\sum_{[b,e] \text{ interesting int.}} (e-b+1)}, \text{ where } var([b,e]) = \text{max no. runs in } [b,e]$$

(depends on Parikh vector)

Why does it matter?

Theoretician: You are all using different methods to compute the BWT of string collections, and the results are pretty different!

Theoretician: You are all using different methods to compute the BWT of string collections, and the results are pretty different!

Programmer: It doesn't matter, all I care about is that it's efficient.

Theoretician: You are all using different methods to compute the BWT of string collections, and the results are pretty different!

Programmer: It doesn't matter, all I care about is that it's efficient.

Theoretician: ... and correct?

Theoretician: You are all using different methods to compute the BWT of string collections, and the results are pretty different!

Programmer: It doesn't matter, all I care about is that it's efficient.

Theoretician: . . . and correct?

Programmer: Ok, but you said yourself that it was all correct!

Theoretician: You are all using different methods to compute the BWT of string collections, and the results are pretty different!

Programmer: It doesn't matter, all I care about is that it's efficient.

Theoretician: . . . and correct?

Programmer: Ok, but you said yourself that it was all correct!

Theoretician: But it's not nice that your tool computes a different thing from your competitor's.

Theoretician: You are all using different methods to compute the BWT of string collections, and the results are pretty different!

Programmer: It doesn't matter, all I care about is that it's efficient.

Theoretician: . . . and correct?

Programmer: Ok, but you said yourself that it was all correct!

Theoretician: But it's not nice that your tool computes a different thing from your competitor's.

Programmer: I am never going to use her tool anyway!

Why you should care

1. number of runs
2. the parameter r is not well-defined
3. input order dependence

1. Number of runs

r = number of runs of the BWT.

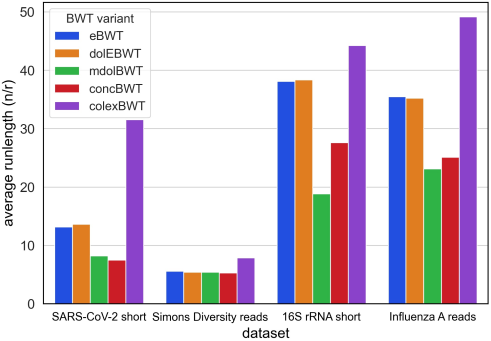
1. Number of runs

r = number of runs of the BWT.

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

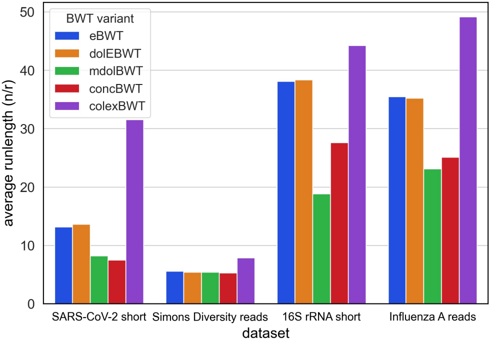
BWT variant	example	r	r w/o \$'s
non-sep. based			
eBWT($\mathcal{M}$)	CGGGATGTACGTTAAAAA	11	11
separator-based			
dollarEBWT($\mathcal{M}$)	GGAAACGG\$\$\$TTACTGT\$AAA\$	14	11
multidolBWT($\mathcal{M}$)	GAGAAGCG\$\$\$TTATCTG\$AAA\$	17	14
colexBWT($\mathcal{M}$)	AAAGGCGG\$\$\$TTACTGT\$AAA\$	14	11
concatBWT($\mathcal{M}$)	AAGAGGGC\$\$\$TTACTGT\$AAA\$	15	12

1. Number of runs



Average runlength (n/r) on four short sequence datasets, of all BWT variants. (500,000 sequences each, of length between 50 and 301.)

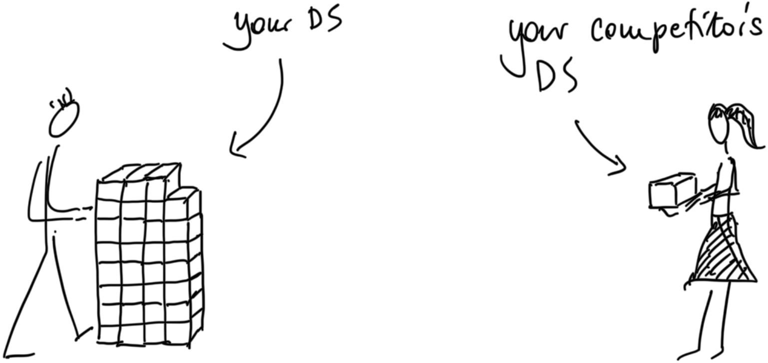
1. Number of runs



Average runlength (n/r) on four short sequence datasets, of all BWT variants.
(500,000 sequences each, of length between 50 and 301.)

- On these datasets, difference of a factor of up to 4.2.
- In a separate work, difference of a factor of up to 31.
  [Cenzato, Guerrini, L., Rosone, DCC 2023]

size of data structures $\mathcal{O}(r)$



So maybe you should care...

2. The parameter r

- size of data structures $\mathcal{O}(r)$ (r-index) Gagie et al. [JACM 2020], Bannai et al. [TCS 2020]

2. The parameter r

- size of data structures $\mathcal{O}(r)$ (r-index) Gagie et al. [JACM 2020], Bannai et al. [TCS 2020]
- algorithms' running time ideally a function of r (not of $n = |T|$)

2. The parameter r

- size of data structures $\mathcal{O}(r)$ (r-index) — Gagie et al. [JACM 2020], Bannai et al. [TCS 2020]
- algorithms' running time ideally a function of r (not of $n = |T|$)
- increasingly used as a repetitiveness measure of T, similar to z (number of Lempel-Ziv phrases)
  - as a property of the dataset — Bannai et al. [TCS 2020], Boucher et al. [ALENEX 2021], Cobas et al. [CPM 2021]

2. The parameter r

- size of data structures $\mathcal{O}(r)$ (r-index) — Gagie et al. [JACM 2020], Bannai et al. [TCS 2020]
- algorithms' running time ideally a function of r (not of $n = |T|$)
- increasingly used as a repetitiveness measure of T, similar to z (number of Lempel-Ziv phrases)
  - as a property of the dataset — Bannai et al. [TCS 2020], Boucher et al. [ALENEX 2021], Cobas et al. [CPM 2021]
  - in theoretical work on repetitiveness measures — Kempa and Kociumaka [FOCS 2020], Navarro [ACM Comp. Surv., 2021], Akagi et al. [Inf. Comp. 2023]

3. Input order dependence

3. Input order dependence

N.B. multidolBWT and concatBWT depend on the input order!

$\mathcal{M}_1$ = [ATATG,TGA,ACG,ATCA,GGA]   mdolBWT($\mathcal{M}_1$) = GAGAAGCG$$$TTATCTGAAA

$\mathcal{M}_2$ = [ACG,ATATG,GGA,TGA,ATCA]   mdolBWT($\mathcal{M}_2$) = GGAAAGGC$$$TTACTGTAAA

$\mathcal{M}_1$ = [ATATG,TGA,ACG,ATCA,GGA]   concBWT($\mathcal{M}_1$) = AAGAGGGC$$$TTACTGTAAA

$\mathcal{M}_2$ = [ACG,ATATG,GGA,TGA,ATCA]   concBWT($\mathcal{M}_2$) = AGAGACGG$$$TTACTTGAAA

3. Input order dependence

N.B. multidolBWT and concatBWT depend on the input order!

$\mathcal{M}_1$ = [ATATG,TGA,ACG,ATCA,GGA] mdolBWT($\mathcal{M}_1$) = GAGAAGCG$$$TTATCTGAAA

$\mathcal{M}_2$ = [ACG,ATATG,GGA,TGA,ATCA] mdolBWT($\mathcal{M}_2$) = GGAAAGGC$$$TTACTGTAAA

$\mathcal{M}_1$ = [ATATG,TGA,ACG,ATCA,GGA] concBWT($\mathcal{M}_1$) = AAGAGGGC$$$TTACTGTAAA

$\mathcal{M}_2$ = [ACG,ATATG,GGA,TGA,ATCA] concBWT($\mathcal{M}_2$) = AGAGACGG$$$TTACTTGAAA

Thus, giving the same dataset to the same tool but in different order can produce very different results! (incl. the number of runs)

The multidollar BWT can simulate all others

Prop. Let $\mathcal{M}$ be given, and L some separator-based BWT on $\mathcal{M}$. Then there exists an input permutation π such that $\text{multidol}(\pi(\mathcal{M})) = L$.

The multidollar BWT can simulate all others

Prop. Let $\mathcal{M}$ be given, and L some separator-based BWT on $\mathcal{M}$. Then there exists an input permutation π such that $\text{multidol}(\pi(\mathcal{M})) = L$.

Proof sketch: colexBWT: colex order, dollarEBWT: lex order, concatBWT: lex order of subseq strings

The multidollar BWT can simulate all others

Prop. Let $\mathcal{M}$ be given, and L some separator-based BWT on $\mathcal{M}$. Then there exists an input permutation π such that multidol$(\pi(\mathcal{M})) = L$.

Proof sketch: colexBWT: colex order, dollarEBWT: lex order, concatBWT: lex order of subseq strings

- Prop. $\Longrightarrow$ any separator-based BWT variant can be computed using the multidollar method

The multidollar BWT can simulate all others

Prop. Let $\mathcal{M}$ be given, and L some separator-based BWT on $\mathcal{M}$. Then there exists an input permutation π such that multidol$(\pi(\mathcal{M})) = L$.

Proof sketch: colexBWT: colex order, dollarEBWT: lex order, concatBWT: lex order of subseq strings

- Prop. $\implies$ any separator-based BWT variant can be computed using the multidollar method
- Bentley, Gibney, and Thankachan [ESA 2020] gave a linear-time algorithm for the input order of multidollar BWT with minimum r

The multidollar BWT can simulate all others

Prop. Let $\mathcal{M}$ be given, and L some separator-based BWT on $\mathcal{M}$. Then there exists an input permutation π such that multidol$(\pi(\mathcal{M})) = L$.

Proof sketch: colexBWT: colex order, dollarEBWT: lex order, concatBWT: lex order of subseq strings

- Prop. $\implies$ any separator-based BWT variant can be computed using the multidollar method
- Bentley, Gibney, and Thankachan [ESA 2020] gave a linear-time algorithm for the input order of multidollar BWT with minimum r
- We implemented this algorithm in our tool `optimalBWT` [Cenzato, Guerrini, L., Rosone, DCC 2023]

Conclusions

Conclusions

- there are different ways of computing the BWT of a string collection
- these are non-equivalent
- the most commonly used ones are input-order dependent
- the number of runs r varies significantly

Conclusions

- there are different ways of computing the BWT of a string collection
- these are non-equivalent
- the most commonly used ones are input-order dependent
- the number of runs r varies significantly
- $\Longrightarrow$ different tools on the same dataset can produce different size data structures

Conclusions

- there are different ways of computing the BWT of a string collection
- these are non-equivalent
- the most commonly used ones are input-order dependent
- the number of runs r varies significantly
- $\Longrightarrow$ different tools on the same dataset can produce different size data structures
- $\Longrightarrow$ the same tool on the same dataset can produce different size data structures

Conclusions

- there are different ways of computing the BWT of a string collection
- these are non-equivalent
- the most commonly used ones are input-order dependent
- the number of runs r varies significantly
- $\implies$ different tools on the same dataset can produce different size data structures
- $\implies$ the same tool on the same dataset can produce different size data structures
- optBWT minimizes r, and has been implemented

Conclusions

- there are different ways of computing the BWT of a string collection
- these are non-equivalent
- the most commonly used ones are input-order dependent
- the number of runs r varies significantly
- $\Longrightarrow$ different tools on the same dataset can produce different size data structures
- $\Longrightarrow$ the same tool on the same dataset can produce different size data structures
- optBWT minimizes r, and has been implemented
- definition of r should be standardized (optBWT or colexBWT)

Open Problems

- upper bound on differences between separator-based BWT variants
- characterize string collections for which differences highest
- analyze differences between eBWT and separator-based BWTs

Open Problems

- upper bound on differences between separator-based BWT variants
- characterize string collections for which differences highest
- analyze differences between eBWT and separator-based BWTs

My personal conclusion:

Definitions matter!

Acknowledgements

- Davide Cenzato and Zsuzsanna Lipták: *A survey of BWT variants for string collections*, `arXiv:2202.13235` (conf. version: CPM 2022)
  `github.com/davidecenzato/BWT-variants-for-string-collections`
- Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone: *Computing the optimal BWT for very large string collections*, DCC 2023.
  `github.com/davidecenzato/optimalBWT`

Acknowledgements

- Davide Cenzato and Zsuzsanna Lipták: *A survey of BWT variants for string collections*, `arXiv:2202.13235` (conf. version: CPM 2022)
  `github.com/davidecenzato/BWT-variants-for-string-collections`
- Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone: *Computing the optimal BWT for very large string collections*, DCC 2023.
  `github.com/davidecenzato/optimalBWT`

Thank you for your attention!