Computing Burrows-Wheeler Similarity Distributions for String Collections

Felipe A. Louza[1] Guilherme P. Telles[2] Simon Gog[3] Zhao Liang [1]

[1]Department of Computing and Mathematics
University of São Paulo, Brazil

[2]Instituto de Computação
Universidade Estadual de Campinas, Brazil

[3]eBay Inc., USA

October 9, 2018

[1]Supported by the grant #2017/09105-0 from the São Paulo Research Foundation (FAPESP).

Outline

1. Introduction

2. Burrows-Wheeler Similarity Distribution (BWSD)

3. Algorithm 1

4. Algorithm 2

5. Experiments

6. References

Introduction

Burrows-Wheeler transform (BWT):

- The BWT is a reversible transformation of a string $T[1, n]$ that tends to group identical symbols into runs.
- BWT(T) is a more compressible string.
- Important to data compression, text indexing, and other applications.

$$\texttt{abracadabra\$} \xrightarrow{BWT} \texttt{ard\$rcaaaabb}$$

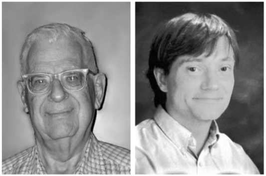


Figure: David Wheeler - Michael Burrows (1994)

Introduction

Burrows-Wheeler transform (BWT):

- The BWT is a reversible transformation of a string $T[1, n]$ that tends to group identical symbols into runs.
- BWT(T) is a more compressible string.
- Important to data compression, text indexing, and other applications.

$$\texttt{abracadabra\$} \xrightarrow{BWT} \texttt{ard\$rcaaaabb}$$

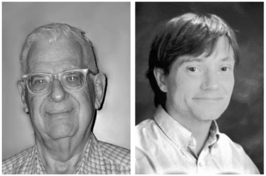


Figure: David Wheeler - Michael Burrows (1994)

Introduction

Burrows-Wheeler transform (BWT):

- The BWT is a reversible transformation of a string $T[1, n]$ that tends to group identical symbols into runs.
- BWT(T) is a more compressible string.
- Important to data compression, text indexing, and other applications.

$$\texttt{abracadabra\$} \quad \xrightarrow{BWT} \quad \texttt{ard\$rcaaaabb}$$

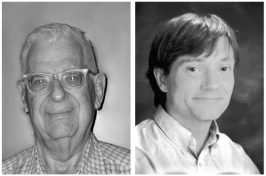


Figure: David Wheeler - Michael Burrows (1994)

Introduction

Burrows-Wheeler transform (BWT):

- The BWT(T) can be obtained by sorting all rotations of $T[1, n]$.
- Taking the last column L=BWT.
- We assume $T[1, n]$ always ends with a terminator symbol $\$ < c \in \Sigma$.

$$\texttt{abracadabra\$} \quad \xrightarrow{BWT} \quad \texttt{ard\$rcaaaabb}$$

Introduction

Burrows-Wheeler transform (BWT):

- The BWT(T) can be obtained by sorting all rotations of $T[1,n]$.
- Taking the last column L=BWT.
- We assume $T[1,n]$ always ends with a terminator symbol $\$ < c \in \Sigma$.

abracadabra$ $\xrightarrow{BWT}$ ard$rcaaaabb

abracadabra$
bracadabra$a
racadabra$ab
acadabra$abr
cadabra$abra
adabra$abrac
dabra$abraca
abra$abracad
bra$abracada
ra$abracadab
a$abracadabr
$abracadabra

M'

Introduction

Burrows-Wheeler transform (BWT):

- The BWT(T) can be obtained by sorting all rotations of $T[1, n]$.
- Taking the last column L=BWT.
- We assume $T[1, n]$ always ends with a terminator symbol $\$ < c \in \Sigma$.

abracadabra\$ $\xrightarrow{BWT}$ ard\$rcaaaabb

```
abracadabra$
bracadabra$a
racadabra$ab
acadabra$abr
cadabra$abra
adabra$abrac
dabra$abraca
abra$abracad
bra$abracada
ra$abracadab
a$abracadabr
$abracadabra
```

$\xrightarrow{sort}$

M'

Introduction

Burrows-Wheeler transform (BWT):

- The BWT(T) can be obtained by sorting all rotations of $T[1, n]$.
- Taking the last column L=BWT.
- We assume $T[1, n]$ always ends with a terminator symbol $\$ < c \in \Sigma$.

`abracadabra$` $\xrightarrow{BWT}$ `ard$rcaaaabb`

		`F          L`
`abracadabra$`		`$abracadabra`
`bracadabra$a`		`a$abracadabr`
`racadabra$ab`		`abra$abracad`
`acadabra$abr`		`abracadabra$`
`cadabra$abra`		`acadabra$abr`
`adabra$abrac`		`adabra$abrac`
`dabra$abraca`	$\xrightarrow{sort}$	`bra$abracada`
`abra$abracad`		`bracadabra$a`
`bra$abracada`		`cadabra$abra`
`ra$abracadab`		`dabra$abraca`
`a$abracadabr`		`ra$abracadab`
`$abracadabra`		`racadabra$ab`
M'		M

Introduction

Burrows-Wheeler transform (BWT):

- The BWT(T) can be obtained by sorting all rotations of $T[1, n]$.
- Taking the last column L=BWT.
- We assume $T[1, n]$ always ends with a terminator symbol $\$ < c \in \Sigma$.

abracadabra$ $\xrightarrow{BWT}$ ard$rcaaaabb

M'	$\xrightarrow{sort}$	M (F … L)
abracadabra$		$abracadabra
bracadabra$a		a$abracadabr
racadabra$ab		abra$abracad
acadabra$abr		abracadabra$
cadabra$abra		acadabra$abr
adabra$abrac		adabra$abrac
dabra$abraca		bra$abracada
abra$abracad		bracadabra$a
bra$abracada		cadabra$abra
ra$abracadab		dabra$abraca
a$abracadabr		ra$abracadab
$abracadabra		racadabra$ab

M' M

Introduction

Burrows-Wheeler transform (BWT):

- The BWT(T) can be obtained by sorting all rotations of $T[1, n]$.
- Taking the last column L=BWT.
- We assume $T[1, n]$ always ends with a terminator symbol $\$ < c \in \Sigma$.

abracadabra\$ $\xrightarrow{BWT}$ ard\$rcaaaabb

M'		M (F … L)
abracadabra\$		\$abracadabra
bracadabra\$a		a\$abracadabr
racadabra\$ab		abra\$abracad
acadabra\$abr		abracadabra\$
cadabra\$abra		acadabra\$abr
adabra\$abrac		adabra\$abrac
dabra\$abraca	$\xrightarrow{sort}$	bra\$abracada
abra\$abracad		bracadabra\$a
bra\$abracada		cadabra\$abra
ra\$abracadab		dabra\$abraca
a\$abracadabr		ra\$abracadab
\$abracadabra		racadabra\$ab

M' M

Sorting all rotations ≈ sorting all sufixes ⇒ no comparison will exceed \$.

Introduction

Burrows-Wheeler transform (BWT):

- The BWT(T) can be obtained by sorting all rotations of $T[1, n]$.
- Taking the last column L=BWT.
- We assume $T[1, n]$ always ends with a terminator symbol $\$ < c \in \Sigma$.

abracadabra\$ $\xrightarrow{BWT}$ ard\$rcaaaabb

M'		M (F … L)
abracadabra\$		\$abracadabra
bracadabra\$a		a\$abracadabr
racadabra\$ab		abra\$abracad
acadabra\$abr		abracadabra\$
cadabra\$abra		acadabra\$abr
adabra\$abrac		adabra\$abrac
dabra\$abraca	$\xrightarrow{sort}$	bra\$abracada
abra\$abracad		bracadabra\$a
bra\$abracada		cadabra\$abra
ra\$abracadab		dabra\$abraca
a\$abracadabr		ra\$abracadab
\$abracadabra		racadabra\$ab

M' M

In practice, we sort all suffixes (Suffix Array) ⇒ take the preceding symbols as BWT.

BWT for multiple strings:

- The BWT can be defined for multiple strings $T_1, T_2, \ldots, T_d$.
  - Concatenate all strings: $T^{cat} = T_1 \cdot T_2 \cdots T_d$, of length N.
  - Each T_i is terminated by a distinct symbol $\$_i$, with $\$_1 < \$_2 < \cdots < \$_d$.
- Compute SA for T^{cat} → BWT
- Document array (DA) gives the string *id* of each BWT symbol.

BWT(banana$\$_1$anaba$\$_2$)

i	DA	BWT	suffixes
1	1	a	$\$_1$
2	2	a	$\$_2$
3	1	n	a$\$_1$
4	2	b	a$\$_2$
5	2	n	aba$\$_2$
6	1	n	ana$\$_1$
7	2	$\$_1$	anaba$\$_2$
8	1	b	anana$\$_1$
9	2	a	ba$\$_2$
10	1	$\$_2$	banana$\$_1$
11	1	a	na$\$_1$
12	2	a	naba$\$_2$
13	1	a	nana$\$_1$

Introduction

BWT for multiple strings:

- The BWT can be defined for multiple strings $T_1, T_2, \ldots, T_d$.
  - Concatenate all strings: $T^{cat} = T_1 \cdot T_2 \cdots T_d$, of length N.
  - Each T_i is terminated by a distinct symbol $\$_i$, with $\$_1 < \$_2 < \cdots < \$_d$.
- Compute SA for T^{cat} $\rightarrow$ BWT
- Document array (DA) gives the string *id* of each BWT symbol.

BWT(banana$\$_1$anaba$\$_2$)

i	DA	BWT	suffixes
1	1	a	$\$_1$
2	2	a	$\$_2$
3	1	n	a$\$_1$
4	2	b	a$\$_2$
5	2	n	aba$\$_2$
6	1	n	ana$\$_1$
7	2	$\$_1$	anaba$\$_2$
8	1	b	anana$\$_1$
9	2	a	ba$\$_2$
10	1	$\$_2$	banana$\$_1$
11	1	a	na$\$_1$
12	2	a	naba$\$_2$
13	1	a	nana$\$_1$

Introduction

BWT for multiple strings:

- The BWT can be defined for multiple strings $T_1, T_2, \ldots, T_d$.
  - Concatenate all strings: $T^{cat} = T_1 \cdot T_2 \cdots T_d$, of length N.
  - Each T_i is terminated by a distinct symbol $\$_i$, with $\$_1 < \$_2 < \cdots < \$_d$.
- Compute SA for T^{cat} $\rightarrow$ BWT
- Document array (DA) gives the string *id* of each BWT symbol.

BWT(banana$\$_1$anaba$\$_2$)

i	DA	BWT	suffixes
1	1	a	$\$_1$
2	2	a	$\$_2$
3	1	n	a$\$_1$
4	2	b	a$\$_2$
5	2	n	aba$\$_2$
6	1	n	ana$\$_1$
7	2	$\$_1$	anaba$\$_2$
8	1	b	anana$\$_1$
9	2	a	ba$\$_2$
10	1	$\$_2$	banana$\$_1$
11	1	a	na$\$_1$
12	2	a	naba$\$_2$
13	1	a	nana$\$_1$

We replace $T_i\$$ by $T_i\$_i$

Introduction

BWT for multiple strings:

- The BWT can be defined for multiple strings $T_1, T_2, \ldots, T_d$.
  - Concatenate all strings: $T^{cat} = T_1 \cdot T_2 \cdots T_d$, of length N.
  - Each T_i is terminated by a distinct symbol $\$_i$, with $\$_1 < \$_2 < \cdots < \$_d$.
- Compute SA for T^{cat} $\rightarrow$ BWT
- Document array (DA) gives the string *id* of each BWT symbol.

BWT(banana$\$_1$anaba$\$_2$)

i	DA	BWT	suffixes
1	1	a	$\$_1$
2	2	a	$\$_2$
3	1	n	a$\$_1$
4	2	b	a$\$_2$
5	2	n	aba$\$_2$
6	1	n	ana$\$_1$
7	2	$\$_1$	anaba$\$_2$
8	1	b	anana$\$_1$
9	2	a	ba$\$_2$
10	1	$\$_2$	banana$\$_1$
11	1	a	na$\$_1$
12	2	a	naba$\$_2$
13	1	a	nana$\$_1$

Introduction

BWT for multiple strings:

- The BWT can be defined for multiple strings $T_1, T_2, \ldots, T_d$.
  - Concatenate all strings: $T^{cat} = T_1 \cdot T_2 \cdots T_d$, of length N.
  - Each T_i is terminated by a distinct symbol $\$_i$, with $\$_1 < \$_2 < \cdots < \$_d$.
- Compute SA for T^{cat} $\rightarrow$ BWT
- Document array (DA) gives the string *id* of each BWT symbol.

BWT(banana$\$_1$anaba$\$_2$)

i	DA	BWT	suffixes
1	1	a	$\$_1$
2	2	a	$\$_2$
3	1	n	a$\$_1$
4	2	b	a$\$_2$
5	2	n	aba$\$_2$
6	1	n	ana$\$_1$
7	2	$\$_1$	anaba$\$_2$
8	1	b	anana$\$_1$
9	2	a	ba$\$_2$
10	1	$\$_2$	banana$\$_1$
11	1	a	na$\$_1$
12	2	a	naba$\$_2$
13	1	a	nana$\$_1$

$DA[i] = 1$ $\rightarrow$ BWT-symbol preceeds suffix from T_1

Outline

1. Introduction

2. Burrows-Wheeler Similarity Distribution (BWSD)

3. Algorithm 1

4. Algorithm 2

5. Experiments

6. References

BWSD

Comparing two strings T_1, T_2 using BWT($T_1 \cdot T_2$):

- Key idea: the more the symbols are intermixed in BWT($T_1 \cdot T_2$) $\Rightarrow$ the greater the number of shared substrings $\Rightarrow$ the more similar T_1 and T_2
- First proposed by Mantaci *et al.* [MRRS08]

BWT(banana\$)

i	BWT	suffixes
1	a	\$
2	n	a\$
3	n	ana\$
4	b	anana\$
5	\$	banana\$
6	a	na\$
7	a	nana\$

BWT(anaba\$)

i	BWT	suffixes
1	a	\$
2	b	a\$
3	n	aba\$
4	\$	anaba\$
5	a	ba\$
6	a	naba\$

BWT(banana$\$_1$anaba$\$_2$)

i	DA	BWT	suffixes
1	1	a	$\$_1$
2	2	a	$\$_2$
3	1	n	a$\$_1$
4	2	b	a$\$_2$
5	2	n	aba$\$_2$
6	1	n	ana$\$_1$
7	2	$\$_1$	anaba$\$_2$
8	1	b	anana$\$_1$
9	2	a	ba$\$_2$
10	1	$\$_2$	banana$\$_1$
11	1	a	na$\$_1$
12	2	a	naba$\$_2$
13	1	a	nana$\$_1$

BWSD

Comparing two strings T_1, T_2 using BWT($T_1 \cdot T_2$):

- Key idea: the more the symbols are intermixed in BWT($T_1 \cdot T_2$) $\Rightarrow$ the greater the number of shared substrings $\Rightarrow$ the more similar T_1 and T_2 are.
- First proposed by Mantaci *et al.* [MRRS08]

BWT(banana\$)

i	BWT	suffixes
1	a	\$
2	n	a\$
3	n	ana\$
4	b	anana\$
5	\$	banana\$
6	a	na\$
7	a	nana\$

BWT(anaba\$)

i	BWT	suffixes
1	a	\$
2	b	a\$
3	n	aba\$
4	\$	anaba\$
5	a	ba\$
6	a	naba\$

BWT(banana$\$_1$anaba$\$_2$)

i	DA	BWT	suffixes
1	1	a	$\$_1$
2	2	a	$\$_2$
3	1	n	a$\$_1$
4	2	b	a$\$_2$
5	2	n	aba$\$_2$
6	1	n	ana$\$_1$
7	2	$\$_1$	anaba$\$_2$
8	1	b	anana$\$_1$
9	2	a	ba$\$_2$
10	1	$\$_2$	banana$\$_1$
11	1	a	na$\$_1$
12	2	a	naba$\$_2$
13	1	a	nana$\$_1$

BWSD

Comparing two strings T_1, T_2 using BWT($T_1 \cdot T_2$):

- Key idea: the more the symbols are intermixed in BWT($T_1 \cdot T_2$) $\Rightarrow$ the greater the number of shared substrings $\Rightarrow$ the more similar T_1 and T_2 are.
- First proposed by Mantaci *et al.* [MRRS08]

BWT(banana\$)

i	BWT	suffixes
1	a	\$
2	n	a\$
3	n	ana\$
4	b	anana\$
5	\$	banana\$
6	a	na\$
7	a	nana\$

BWT(anaba\$)

i	BWT	suffixes
1	a	\$
2	b	a\$
3	n	aba\$
4	\$	anaba\$
5	a	ba\$
6	a	naba\$

BWT(banana$\$_1$anaba$\$_2$)

i	DA	BWT	suffixes
1	1	a	$\$_1$
2	2	a	$\$_2$
3	1	n	a$\$_1$
4	2	b	a$\$_2$
5	2	n	aba$\$_2$
6	1	n	ana$\$_1$
7	2	$\$_1$	anaba$\$_2$
8	1	b	anana$\$_1$
9	2	a	ba$\$_2$
10	1	$\$_2$	banana$\$_1$
11	1	a	na$\$_1$
12	2	a	naba$\$_2$
13	1	a	nana$\$_1$

BWSD

Comparing two strings T_1, T_2 using BWT($T_1 \cdot T_2$):

- Key idea: the more the symbols are intermixed in BWT($T_1 \cdot T_2$) $\Rightarrow$ the greater the number of shared substrings $\Rightarrow$ the more similar T_1 and T_2 are.
- First proposed by Mantaci *et al.* [MRRS08]

BWT(banana\$)

i	BWT	suffixes
1	a	\$
2	n	a\$
3	n	ana\$
4	b	anana\$
5	\$	banana\$
6	a	na\$
7	a	nana\$

BWT(anaba\$)

i	BWT	suffixes
1	a	\$
2	b	a\$
3	n	aba\$
4	\$	anaba\$
5	a	ba\$
6	a	naba\$

BWT(banana$\$_1$anaba$\$_2$)

i	DA	BWT	suffixes
1	1	a	$\$_1$
2	2	a	$\$_2$
3	1	n	a$\$_1$
4	2	b	a$\$_2$
5	2	n	aba$\$_2$
6	1	n	ana$\$_1$
7	2	$\$_1$	anaba$\$_2$
8	1	b	anana$\$_1$
9	2	a	ba$\$_2$
10	1	$\$_2$	banana$\$_1$
11	1	a	na$\$_1$
12	2	a	naba$\$_2$
13	1	a	nana$\$_1$

BWSD

Comparing two strings T_1, T_2 using BWT($T_1 \cdot T_2$):

- Key idea: the more the symbols are intermixed in BWT($T_1 \cdot T_2$) $\Rightarrow$ the greater the number of shared substrings $\Rightarrow$ the more similar T_1 and T_2 are.
- First proposed by Mantaci *et al.* [MRRS08]

BWT(banana\$)

i	BWT	suffixes
1	a	\$
2	n	a\$
3	n	ana\$
4	b	anana\$
5	\$	banana\$
6	a	na\$
7	a	nana\$

BWT(anaba\$)

i	BWT	suffixes
1	a	\$
2	b	a\$
3	n	aba\$
4	\$	anaba\$
5	a	ba\$
6	a	naba\$

BWT(banana$\$_1$anaba$\$_2$)

i	DA	BWT	suffixes
1	1	a	$\$_1$
2	2	a	$\$_2$
3	1	n	a$\$_1$
4	2	b	a$\$_2$
5	2	n	aba$\$_2$
6	1	n	ana$\$_1$
7	2	$\$_1$	anaba$\$_2$
8	1	b	anana$\$_1$
9	2	a	ba$\$_2$
10	1	$\$_2$	banana$\$_1$
11	1	a	na$\$_1$
12	2	a	naba$\$_2$
13	1	a	nana$\$_1$

BWSD

Yang *et al.* [YZW10] introduced the Burrows-Wheeler Similarity Distribution (BWSD):

- The BWSD(T_1, T_2) is constructed as follows:

1. Create a bitvector $\alpha_{1,2}$, such that $\alpha_{1,2}[i] = 0$ if DA$[i] = 1$, $\alpha_{1,2}[i] = 1$ otherwise;

$$\mathrm{BWT}(T_1 T_2) = \mathtt{aanbnn\$_1ba\$_2aaa}$$

$$\alpha_{1,2} = \{0, 1, 0, \underline{1, 1}, 0, 1, 0, 1, \underline{0, 0}, 1, 0\}$$

2. Re-write $\alpha_{1,2}$ in the form:

$$r_{1,2} = 0^1 1^1 0^1 \underline{1^2} 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 0^1 1^0$$

3. Count t_{k_j} be the number of runs 0^{k_j} and 1^{k_j} in r:

$$t_1 = 9, t_2 = 2$$

4. Compute sum of all terms: $s = t_1 + t_2 + \ldots + t_{k_j} + \ldots + t_{k_{\max}}$.

$$s = 11$$

BWSD

Yang *et al.* [YZW10] introduced the Burrows-Wheeler Similarity Distribution (BWSD):

- The BWSD(T_1, T_2) is constructed as follows:

1. Create a bitvector $\alpha_{1,2}$, such that $\alpha_{1,2}[i] = 0$ if DA$[i] = 1$, $\alpha_{1,2}[i] = 1$ otherwise;

$$\mathrm{BWT}(T_1 T_2) = \mathtt{aanbnn\$_1ba\$_2aaa}$$

$$\alpha_{1,2} = \{0, 1, 0, \underline{1, 1}, 0, 1, 0, 1, \underline{0, 0}, 1, 0\}$$

2. Re-write $\alpha_{1,2}$ in the form:

$$r_{1,2} = 0^1 1^1 0^1 \underline{1^2} 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 0^1 1^0$$

3. Count t_{k_j} be the number of runs 0^{k_j} and 1^{k_j} in r:

$$t_1 = 9, t_2 = 2$$

4. Compute sum of all terms: $s = t_1 + t_2 + \ldots + t_{k_j} + \ldots + t_{k_{max}}$.

$$s = 11$$

BWSD

Yang *et al.* [YZW10] introduced the Burrows-Wheeler Similarity Distribution (BWSD):

- The BWSD(T_1, T_2) is constructed as follows:

1. Create a bitvector $\alpha_{1,2}$, such that $\alpha_{1,2}[i] = 0$ if DA$[i] = 1$, $\alpha_{1,2}[i] = 1$ otherwise;

$$\text{BWT}(T_1 T_2) = \texttt{aanbnn\$}_1\texttt{ba\$}_2\texttt{aaa}$$

$$\alpha_{1,2} = \{0, 1, 0, \underline{1, 1}, 0, 1, 0, 1, \underline{0, 0}, 1, 0\}$$

2. Re-write $\alpha_{1,2}$ in the form:

$$r_{1,2} = 0^1 1^1 0^1 \underline{1^2} 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 0^1 1^0$$

3. Count t_{k_j} be the number of runs 0^{k_j} and 1^{k_j} in r:

$$t_1 = 9, t_2 = 2$$

4. Compute sum of all terms: $s = t_1 + t_2 + \ldots + t_{k_j} + \ldots + t_{k_{\max}}$.

$$s = 11$$

BWSD

Yang *et al.* [YZW10] introduced the Burrows-Wheeler Similarity Distribution (BWSD):

- The BWSD(T_1, T_2) is constructed as follows:

1. Create a bitvector $\alpha_{1,2}$, such that $\alpha_{1,2}[i] = 0$ if DA$[i] = 1$, $\alpha_{1,2}[i] = 1$ otherwise;

$$\text{BWT}(T_1 T_2) = \texttt{aanbnn\$}_1\texttt{ba\$}_2\texttt{aaa}$$

$$\alpha_{1,2} = \{0, 1, 0, \underline{1, 1}, 0, 1, 0, 1, \underline{0, 0}, 1, 0\}$$

2. Re-write $\alpha_{1,2}$ in the form:

$$r_{1,2} = 0^1 1^1 0^1 \underline{1^2} 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 0^1 1^0$$

3. Count t_{k_j} be the number of runs 0^{k_j} and 1^{k_j} in r:

$$t_1 = 9, t_2 = 2$$

4. Compute sum of all terms: $s = t_1 + t_2 + \ldots + t_{k_j} + \ldots + t_{k_{\max}}$.

$$s = 11$$

BWSD

Yang *et al.* [YZW10] introduced the Burrows-Wheeler Similarity Distribution (BWSD):

- The BWSD(T_1, T_2) is constructed as follows:

1. Create a bitvector $\alpha_{1,2}$, such that $\alpha_{1,2}[i] = 0$ if DA$[i] = 1$, $\alpha_{1,2}[i] = 1$ otherwise;

$$\text{BWT}(T_1 T_2) = \texttt{aanbnn\$}_1\texttt{ba\$}_2\texttt{aaa}$$

$$\alpha_{1,2} = \{0, 1, 0, \underline{1, 1}, 0, 1, 0, 1, \underline{0, 0}, 1, 0\}$$

2. Re-write $\alpha_{1,2}$ in the form:

$$r_{1,2} = 0^1 1^1 0^1 \underline{1^2} 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 0^1 1^0$$

3. Count t_{k_j} be the number of runs 0^{k_j} and 1^{k_j} in r:

$$t_1 = 9, t_2 = 2$$

4. Compute sum of all terms: $s = t_1 + t_2 + \ldots + t_{k_j} + \ldots + t_{k_{\max}}$.

$$s = 11$$

BWSD

Yang *et al.* [YZW10] introduced the Burrows-Wheeler Similarity Distribution (BWSD):

- The BWSD(T_1, T_2) is a probability mass function:
  - $P\{k_j = k\} = t_k/s$ for $k = 1, 2, \ldots, k_{\text{max}}$.
  - We have:

$$r = 0^1 1^1 0^1 \underline{1^2} 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 0^1 1^0$$

$$t_1 = 9, t_2 = 2 \text{ and } s = 11$$

$$\text{BWSD}(T_1, T_2) \text{ is } \underline{P\{k_j = 1\} = 9/11}, \underline{P\{k_j = 2\} = 2/11}$$

- Yang *et al.* [YZW10] defined two distances based on:
  1. Expectation of BWSD(T_1, T_2).
  2. Shannon entropy of BWSD(T_1, T_2).
- Properties:
  - Symmetric: $\mathrm{D_M}(T_1, T_2) = \mathrm{D_M}(T_2, T_1)$
  - $\mathrm{D_M}(T_1, T_1) = 0$

BWSD

Yang *et al.* [YZW10] introduced the Burrows-Wheeler Similarity Distribution (BWSD):

- The BWSD(T_1, T_2) is a probability mass function:
  - $P\{k_j = k\} = t_k/s$ for $k = 1, 2, \ldots, k_{\text{max}}$.
  - We have:

$$r = 0^1 1^1 0^1 \underline{1^2} 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 0^1 1^0$$

$$t_1 = 9, t_2 = 2 \text{ and } s = 11$$

$$\text{BWSD}(T_1, T_2) \text{ is } \underline{P\{k_j = 1\} = 9/11}, \underline{P\{k_j = 2\} = 2/11}$$

- Yang *et al.* [YZW10] defined two distances based on:
  1. Expectation of BWSD(T_1, T_2).
  2. Shannon entropy of BWSD(T_1, T_2).

- Properties:
  - Symmetric: $\mathrm{D_M}(T_1, T_2) = \mathrm{D_M}(T_2, T_1)$
  - $\mathrm{D_M}(T_1, T_1) = 0$

BWSD

Yang *et al.* [YZW10] introduced the Burrows-Wheeler Similarity Distribution (BWSD):

- The BWSD(T_1, T_2) is a probability mass function:
  - $P\{k_j = k\} = t_k/s$ for $k = 1, 2, \ldots, k_{\max}$.
  - We have:

$$r = 0^1 1^1 0^1 \underline{1^2} 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 0^1 1^0$$

$$t_1 = 9, t_2 = 2 \text{ and } s = 11$$

$$\text{BWSD}(T_1, T_2) \text{ is } \underline{P\{k_j = 1\} = 9/11}, \underline{P\{k_j = 2\} = 2/11}$$

- Yang *et al.* [YZW10] defined two distances based on:
  1. Expectation of BWSD(T_1, T_2).
  2. Shannon entropy of BWSD(T_1, T_2).
- Properties:
  - Symmetric: $\mathrm{D_M}(T_1, T_2) = \mathrm{D_M}(T_2, T_1)$
  - $\mathrm{D_M}(T_1, T_1) = 0$

BWSD

Applications:

- The BWSD was evaluated with the construction of phylogenetic trees [YCZW10].
- A matrix $\mathrm{M}_{d \times d}$ with all pairs of distances[2] is computed $\Rightarrow$ given as input for algorithms like UPGMA and Neighbor-Joining.

0	$\mathrm{D_M}(T_1, T_2)$	$\mathrm{D_M}(T_1, T_3)$	$\mathrm{D_M}(T_1, T_4)$	$\mathrm{D_M}(T_1, T_5)$
	0	$\mathrm{D_M}(T_2, T_3)$	$\mathrm{D_M}(T_2, T_4)$	$\mathrm{D_M}(T_2, T_5)$
		0	$\mathrm{D_M}(T_3, T_4)$	$\mathrm{D_M}(T_3, T_5)$
			0	$\mathrm{D_M}(T_5, T_5)$
				0

$\mathrm{M}_{d \times d}$

$\leftarrow$

0	$\mathrm{BWT}(T_1 T_2)$	$\mathrm{BWT}(T_1 T_3)$	$\mathrm{BWT}(T_1 T_4)$	$\mathrm{BWT}(T_1 T_5)$
	0	$\mathrm{BWT}(T_2 T_3)$	$\mathrm{BWT}(T_2 T_4)$	$\mathrm{BWT}(T_2 T_5)$
		0	$\mathrm{BWT}(T_3 T_4)$	$\mathrm{BWT}(T_3 T_5)$
			0	$\mathrm{BWT}(T_4 T_5)$
				0

BWTs

Straightforward algorithm:

- Compute all pairwise $\mathrm{BWT}(T_i, T_j)$, for all $i, j > i \leftarrow O(dN)$-time.

Our contribution:

- We present 2 algorithms $\leftarrow$ $O(dN)$-time, $O(N + z)$-time

[2] Actually, only upper triangular entries of $\mathrm{M}_{d \times d}$.

BWSD

Applications:

- The BWSD was evaluated with the construction of phylogenetic trees [YCZW10].
- A matrix $\mathrm{M}_{d\times d}$ with all pairs of distances[2] is computed $\Rightarrow$ given as input for algorithms like UPGMA and Neighbor-Joining.

0	$\mathrm{D_M}(T_1, T_2)$	$\mathrm{D_M}(T_1, T_3)$	$\mathrm{D_M}(T_1, T_4)$	$\mathrm{D_M}(T_1, T_5)$
	0	$\mathrm{D_M}(T_2, T_3)$	$\mathrm{D_M}(T_2, T_4)$	$\mathrm{D_M}(T_2, T_5)$
		0	$\mathrm{D_M}(T_3, T_4)$	$\mathrm{D_M}(T_3, T_5)$
			0	$\mathrm{D_M}(T_5, T_5)$
				0

$\mathrm{M}_{d\times d}$

$\leftarrow$

0	$\mathrm{BWT}(T_1 T_2)$	$\mathrm{BWT}(T_1 T_3)$	$\mathrm{BWT}(T_1 T_4)$	$\mathrm{BWT}(T_1 T_5)$
	0	$\mathrm{BWT}(T_2 T_3)$	$\mathrm{BWT}(T_2 T_4)$	$\mathrm{BWT}(T_2 T_5)$
		0	$\mathrm{BWT}(T_3 T_4)$	$\mathrm{BWT}(T_3 T_5)$
			0	$\mathrm{BWT}(T_4 T_5)$
				0

BWTs

Straightforward algorithm:

- Compute all pairwise $\mathrm{BWT}(T_i, T_j)$, for all $i, j > i \leftarrow O(dN)$-time.

Our contribution:

- We present 2 algorithms $\leftarrow O(dN)$-time, $O(N + z)$-time

[2] Actually, only upper triangular entries of $\mathrm{M}_{d\times d}$.

BWSD

Applications:

- The BWSD was evaluated with the construction of phylogenetic trees [YCZW10].
- A matrix $\mathrm{M}_{d\times d}$ with all pairs of distances[2] is computed $\Rightarrow$ given as input for algorithms like UPGMA and Neighbor-Joining.

0	$\mathrm{D_M}(T_1, T_2)$	$\mathrm{D_M}(T_1, T_3)$	$\mathrm{D_M}(T_1, T_4)$	$\mathrm{D_M}(T_1, T_5)$
	0	$\mathrm{D_M}(T_2, T_3)$	$\mathrm{D_M}(T_2, T_4)$	$\mathrm{D_M}(T_2, T_5)$
		0	$\mathrm{D_M}(T_3, T_4)$	$\mathrm{D_M}(T_3, T_5)$
			0	$\mathrm{D_M}(T_5, T_5)$
				0

$\mathrm{M}_{d\times d}$

$\leftarrow$

0	$\mathrm{BWT}(T_1 T_2)$	$\mathrm{BWT}(T_1 T_3)$	$\mathrm{BWT}(T_1 T_4)$	$\mathrm{BWT}(T_1 T_5)$
	0	$\mathrm{BWT}(T_2 T_3)$	$\mathrm{BWT}(T_2 T_4)$	$\mathrm{BWT}(T_2 T_5)$
		0	$\mathrm{BWT}(T_3 T_4)$	$\mathrm{BWT}(T_3 T_5)$
			0	$\mathrm{BWT}(T_4 T_5)$
				0

BWTs

Straightforward algorithm:

- Compute all pairwise $\mathrm{BWT}(T_i, T_j)$, for all $i, j > i \leftarrow O(dN)$-time.

Our contribution:

- We present 2 algorithms $\leftarrow O(dN)$-time, $O(N+z)$-time

[2] Actually, only upper triangular entries of $\mathrm{M}_{d\times d}$.

BWSD

Applications:

- The BWSD was evaluated with the construction of phylogenetic trees [YCZW10].
- A matrix $\mathrm{M}_{d \times d}$ with all pairs of distances[2] is computed $\Rightarrow$ given as input for algorithms like UPGMA and Neighbor-Joining.

0	$\mathrm{D_M}(T_1, T_2)$	$\mathrm{D_M}(T_1, T_3)$	$\mathrm{D_M}(T_1, T_4)$	$\mathrm{D_M}(T_1, T_5)$
	0	$\mathrm{D_M}(T_2, T_3)$	$\mathrm{D_M}(T_2, T_4)$	$\mathrm{D_M}(T_2, T_5)$
		0	$\mathrm{D_M}(T_3, T_4)$	$\mathrm{D_M}(T_3, T_5)$
			0	$\mathrm{D_M}(T_5, T_5)$
				0

$\mathrm{M}_{d \times d}$

$\leftarrow$

0	$\mathrm{BWT}(T_1 T_2)$	$\mathrm{BWT}(T_1 T_3)$	$\mathrm{BWT}(T_1 T_4)$	$\mathrm{BWT}(T_1 T_5)$
	0	$\mathrm{BWT}(T_2 T_3)$	$\mathrm{BWT}(T_2 T_4)$	$\mathrm{BWT}(T_2 T_5)$
		0	$\mathrm{BWT}(T_3 T_4)$	$\mathrm{BWT}(T_3 T_5)$
			0	$\mathrm{BWT}(T_4 T_5)$
				0

BWTs

Straightforward algorithm:

- Compute all pairwise $\mathrm{BWT}(T_i, T_j)$, for all $i, j > i \leftarrow O(dN)$-time.

Our contribution:

- We present 2 algorithms $\leftarrow O(dN)$-time, $O(N + z)$-time

[2] Actually, only upper triangular entries of $\mathrm{M}_{d \times d}$.

BWSD

Applications:

- The BWSD was evaluated with the construction of phylogenetic trees [YCZW10].
- A matrix $\mathrm{M}_{d\times d}$ with all pairs of distances[2] is computed $\Rightarrow$ given as input for algorithms like UPGMA and Neighbor-Joining.

0	$\mathrm{D_M}(T_1, T_2)$	$\mathrm{D_M}(T_1, T_3)$	$\mathrm{D_M}(T_1, T_4)$	$\mathrm{D_M}(T_1, T_5)$
	0	$\mathrm{D_M}(T_2, T_3)$	$\mathrm{D_M}(T_2, T_4)$	$\mathrm{D_M}(T_2, T_5)$
		0	$\mathrm{D_M}(T_3, T_4)$	$\mathrm{D_M}(T_3, T_5)$
			0	$\mathrm{D_M}(T_5, T_5)$
				0

$\mathrm{M}_{d\times d}$

$\leftarrow$

0	$\mathrm{BWT}(T_1T_2)$	$\mathrm{BWT}(T_1T_3)$	$\mathrm{BWT}(T_1T_4)$	$\mathrm{BWT}(T_1T_5)$
	0	$\mathrm{BWT}(T_2T_3)$	$\mathrm{BWT}(T_2T_4)$	$\mathrm{BWT}(T_2T_5)$
		0	$\mathrm{BWT}(T_3T_4)$	$\mathrm{BWT}(T_3T_5)$
			0	$\mathrm{BWT}(T_4T_5)$
				0

BWTs

Straightforward algorithm:

- Compute all pairwise $\mathrm{BWT}(T_i, T_j)$, for all $i, j > i \leftarrow O(dN)$-time.

Our contribution:

- We present 2 algorithms $\leftarrow O(dN)$-time, $O(N+z)$-time

[2] Actually, only upper triangular entries of $\mathrm{M}_{d\times d}$.

Outline

1. Introduction

2. Burrows-Wheeler Similarity Distribution (BWSD)

3. Algorithm 1

4. Algorithm 2

5. Experiments

6. References

Algorithm 1

Steps:

1. Compute BWT and DA for $T^{cat} = T_1 T_2 \ldots T_d$.
2. Build d bitvectors $\mathsf{B}_i[1, N]$, where $\mathsf{B}_i[j] = 1$ if $\mathsf{DA}[j] = i$ with rank/select support.
3. For each string T_i, compute $\mathsf{r}_{i,j}$, with $j > i$:

$T_1 = \texttt{banana\$}$, $T_2 = \texttt{anaba\$}$, $T_3 = \texttt{ba\$}$, $T_4 = \texttt{banana\$}$, $T_5 = \texttt{aba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

Algorithm 1

Steps:

1. Compute BWT and DA for $T^{cat} = T_1 T_2 \dots T_d$.
2. Build d bitvectors $\mathsf{B}_i[1, N]$, where $\mathsf{B}_i[j] = 1$ if $\mathsf{DA}[j] = i$ with rank/select support.
3. For each string T_i, compute $\mathsf{r}_{i,j}$, with $j > i$:
   3.1 Select the intervals $\mathsf{DA}[a, b]$ that contain consecutive entries of i.

$T_1 = \texttt{banana\$}$, $T_2 = \texttt{anaba\$}$, $T_3 = \texttt{ba\$}$, $T_4 = \texttt{banana\$}$, $T_5 = \texttt{aba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Algorithm 1

Steps:

1. Compute BWT and DA for $T^{cat} = T_1 T_2 \ldots T_d$.
2. Build d bitvectors $\mathsf{B}_i[1, N]$, where $\mathsf{B}_i[j] = 1$ if $\mathsf{DA}[j] = i$ with rank/select support.
3. For each string T_i, compute $\mathsf{r}_{i,j}$, with $j > i$:

   3.1 Select the intervals $\mathsf{DA}[a, b]$ that contain consecutive entries of i.

   3.2 Count k_j occurrences of $j \Rightarrow 0^1 1^{k_j}$, $\mathsf{rank}_1(\mathsf{B}_j, b) - \mathsf{rank}_1(\mathsf{B}_j, a)$.

   3.3 Whenever $j \notin \mathsf{DA}[a, b]$, we collapse $0^{\ell_j} 1^0 0^1 \Rightarrow 0^{\ell_j + 1}$.

$T_1 = \mathtt{banana\$}$, $T_2 = \mathtt{anaba\$}$, $T_3 = \mathtt{ba\$}$, $T_4 = \mathtt{banana\$}$, $T_5 = \mathtt{aba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

$\mathsf{r}_{1,2} = 0^1 1^1 0^1 1^2 0^1 1^1 0^1 1^1 0^2 1^1$

$\mathsf{r}_{1,3} = 0^1 1^1 0^1 1^1 0^2 1^1$

$\mathsf{r}_{1,4} = 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1$

$\mathsf{r}_{1,5} = 0^1 1^1 0^1 1^2 0^2 1^1$

Algorithm 1

Steps:

1. Compute BWT and DA for $T^{cat} = T_1 T_2 \dots T_d$.
2. Build d bitvectors $\mathsf{B}_i[1, N]$, where $\mathsf{B}_i[j] = 1$ if $\mathsf{DA}[j] = i$ with rank/select support.
3. For each string T_i, compute $\mathsf{r}_{\mathsf{i,j}}$, with $j > i$:
   3.1 Select the intervals $\mathsf{DA}[a, b]$ that contain consecutive entries of i.
   3.2 Count k_j occurrences of $j \Rightarrow 0^1 1^{k_j}$, $\mathsf{rank}_1(\mathsf{B}_j, b) - \mathsf{rank}_1(\mathsf{B}_j, a)$.
   3.3 Whenever $j \notin \mathsf{DA}[a, b]$, we collapse $0^{\ell_j} 1^0 0^1 \Rightarrow 0^{\ell_j + 1}$.

$T_1 = \texttt{banana\$}$, $T_2 = \texttt{anaba\$}$, $T_3 = \texttt{ba\$}$, $T_4 = \texttt{banana\$}$, $T_5 = \texttt{aba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

$\mathsf{r}_{1,2} = 0^1 1^1 0^1 1^2 0^1 1^1 0^1 1^1 0^2 1^1$

$\mathsf{r}_{1,3} = 0^1 1^1 0^1 1^1 0^2 1^1$

$\mathsf{r}_{1,4} = 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1$

$\mathsf{r}_{1,5} = 0^1 1^1 0^1 1^2 0^2 1^1$

Algorithm 1

Steps:

1. Compute BWT and DA for $T^{cat} = T_1 T_2 \dots T_d$.
2. Build d bitvectors $\mathsf{B}_i[1, N]$, where $\mathsf{B}_i[j] = 1$ if $\mathsf{DA}[j] = i$ with rank/select support.
3. For each string T_i, compute $\mathsf{r}_{i,j}$, with $j > i$:
   3.1 Select the intervals $\mathsf{DA}[a, b]$ that contain consecutive entries of i.
   3.2 Count k_j occurrences of $j \Rightarrow 0^1 1^{k_j}$, $\mathsf{rank}_1(\mathsf{B}_j, b) - \mathsf{rank}_1(\mathsf{B}_j, a)$.
   3.3 Whenever $j \notin \mathsf{DA}[a, b]$, we collapse $0^{\ell_j} 1^0 0^1 \Rightarrow 0^{\ell_j + 1}$.

$T_1 = \texttt{banana\$}$, $T_2 = \texttt{anaba\$}$, $T_3 = \texttt{ba\$}$, $T_4 = \texttt{banana\$}$, $T_5 = \texttt{aba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4
B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

$\mathsf{r}_{1,2} = 0^1 1^1 0^1 1^2 0^1 1^1 0^1 1^1 0^2 1^1$

$\mathsf{r}_{1,3} = 0^1 1^1 0^1 1^1 0^2 1^1$

$\mathsf{r}_{1,4} = 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1$

$\mathsf{r}_{1,5} = 0^1 1^1 0^1 1^2 0^2 1^1$

Algorithm 1

Steps:

1. Compute BWT and DA for $T^{cat} = T_1 T_2 \dots T_d$.
2. Build d bitvectors $\mathsf{B}_i[1, N]$, where $\mathsf{B}_i[j] = 1$ if $\mathsf{DA}[j] = i$ with rank/select support.
3. For each string T_i, compute $\mathsf{r}_{i,j}$, with $j > i$:
   3.1 Select the intervals $\mathsf{DA}[a, b]$ that contain consecutive entries of i.
   3.2 Count k_j occurrences of $j \Rightarrow 0^1 1^{k_j}$, $\mathsf{rank}_1(\mathsf{B}_j, b) - \mathsf{rank}_1(\mathsf{B}_j, a)$.
   3.3 Whenever $j \notin \mathsf{DA}[a, b]$, we collapse $0^{\ell_j} 1^0 0^1 \Rightarrow 0^{\ell_j + 1}$.

$T_1 = \texttt{banana\$}$, $T_2 = \texttt{anaba\$}$, $T_3 = \texttt{ba\$}$, $T_4 = \texttt{banana\$}$, $T_5 = \texttt{aba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

$\mathsf{r}_{1,2} = 0^1 1^1 0^1 1^2 0^1 1^1 0^1 1^1 0^2 1^1$

$\mathsf{r}_{1,3} = 0^1 1^1 0^1 1^1 0^2 1^1$

$\mathsf{r}_{1,4} = 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1$

$\mathsf{r}_{1,5} = 0^1 1^1 0^1 1^2 0^2 1^1$

Algorithm 1

Steps:

1. Compute BWT and DA for $T^{cat} = T_1 T_2 \ldots T_d$.
2. Build d bitvectors $\mathsf{B}_i[1, N]$, where $\mathsf{B}_i[j] = 1$ if $\mathsf{DA}[j] = i$ with rank/select support.
3. For each string T_i, compute $\mathsf{r}_{\mathsf{i,j}}$, with $j > i$:
   - 3.1 Select the intervals $\mathsf{DA}[a, b]$ that contain consecutive entries of i.
   - 3.2 Count k_j occurrences of $j \Rightarrow 0^1 1^{k_j}$, $\mathsf{rank}_1(\mathsf{B}_j, b) - \mathsf{rank}_1(\mathsf{B}_j, a)$.
   - 3.3 Whenever $j \notin \mathsf{DA}[a, b]$, we collapse $0^{\ell_j} 1^0 0^1 \Rightarrow 0^{\ell_j + 1}$.

$T_1 = \texttt{banana\$}$, $T_2 = \texttt{anaba\$}$, $T_3 = \texttt{ba\$}$, $T_4 = \texttt{banana\$}$, $T_5 = \texttt{aba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

$\mathsf{r}_{1,2} = 0^1 1^1 0^1 1^2 0^1 1^1 0^1 1^1 0^2 1^1$

$\mathsf{r}_{1,3} = 0^1 1^1 0^1 1^1 0^1 1^0 0^2 1^1$

$\mathsf{r}_{1,4} = 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1$

$\mathsf{r}_{1,5} = 0^1 1^1 0^1 1^2 0^1 1^0 0^2 1^1$

Algorithm 1

Steps:

1. Compute BWT and DA for $T^{cat} = T_1 T_2 \dots T_d$.
2. Build d bitvectors $\mathsf{B}_i[1, N]$, where $\mathsf{B}_i[j] = 1$ if $\mathsf{DA}[j] = i$ with rank/select support.
3. For each string T_i, compute $\mathsf{r}_{i,j}$, with $j > i$:
   - 3.1 Select the intervals $\mathsf{DA}[a, b]$ that contain consecutive entries of i.
   - 3.2 Count k_j occurrences of $j \Rightarrow 0^1 1^{k_j}$, $\mathsf{rank}_1(\mathsf{B}_j, b) - \mathsf{rank}_1(\mathsf{B}_j, a)$.
   - 3.3 Whenever $j \notin \mathsf{DA}[a, b]$, we collapse $0^{\ell_j} 1^0 0^1 \Rightarrow 0^{\ell_j + 1}$.

$T_1 = \texttt{banana\$}$, $T_2 = \texttt{anaba\$}$, $T_3 = \texttt{ba\$}$, $T_4 = \texttt{banana\$}$, $T_5 = \texttt{aba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

$\mathsf{r}_{1,2} = 0^1 1^1 0^1 1^2 0^1 1^1 0^1 1^1 0^2 1^1$

$\mathsf{r}_{1,3} = 0^1 1^1 0^1 1^1 0^2 1^1$

$\mathsf{r}_{1,4} = 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1$

$\mathsf{r}_{1,5} = 0^1 1^1 0^1 1^2 0^2 1^1$

Algorithm 1

Steps:

1. Compute BWT and DA for $T^{cat} = T_1 T_2 \ldots T_d$.
2. Build d bitvectors $\mathrm{B}_i[1, N]$, where $\mathrm{B}_i[j] = 1$ if $\mathrm{DA}[j] = i$ with rank/select support.
3. For each string T_i, compute $\mathrm{r_{i,j}}$, with $j > i$:
   3.1 Select the intervals $\mathrm{DA}[a, b]$ that contain consecutive entries of i.
   3.2 Count k_j occurrences of $j \Rightarrow 0^1 1^{k_j}$, $\mathrm{rank}_1(\mathrm{B}_j, b) - \mathrm{rank}_1(\mathrm{B}_j, a)$.
   3.3 Whenever $j \notin \mathrm{DA}[a, b]$, we collapse $0^{\ell_j} 1^0 0^1 \Rightarrow 0^{\ell_j + 1}$.

$T_1 = \mathtt{banana\$}$, $T_2 = \mathtt{anaba\$}$, $T_3 = \mathtt{ba\$}$, $T_4 = \mathtt{banana\$}$, $T_5 = \mathtt{aba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

$\mathrm{r}_{1,2} = 0^1 1^1 0^1 1^2 0^1 1^1 0^1 1^1 0^1 1^0 0^2 1^1$

$\mathrm{r}_{1,3} = 0^1 1^1 0^1 1^1 0^2 1^1 0^1 1^0$

$\mathrm{r}_{1,4} = 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1$

$\mathrm{r}_{1,5} = 0^1 1^1 0^1 1^2 0^2 1^1 0^1 1^0$

Algorithm 1

Steps:

1. Compute <u>BWT and DA</u> for $T^{cat} = T_1 T_2 \ldots T_d$.
2. Build <u>d bitvectors</u> $\mathsf{B}_i[1, N]$, where $\mathsf{B}_i[j] = 1$ if $\mathsf{DA}[j] = i$ with <u>rank/select support</u>.
3. For each string T_i, <u>compute $\mathsf{r_{i,j}}$</u>, with $j > i$:
   - 3.1 Select the intervals $\underline{\mathsf{DA}[a, b]}$ that contain consecutive entries of i.
   - 3.2 Count k_j occurrences of $j \Rightarrow \underline{0^1 1^{k_j}}$, $\underline{\mathsf{rank}_1(\mathsf{B}_j, b) - \mathsf{rank}_1(\mathsf{B}_j, a)}$.
   - 3.3 Whenever $\underline{j \notin \mathsf{DA}[a, b]}$, we collapse $\underline{0^{\ell_j} 1^0 0^1} \Rightarrow \underline{0^{\ell_j + 1}}$.

$T_1 = \mathtt{banana\$}$, $T_2 = \mathtt{anaba\$}$, $T_3 = \mathtt{ba\$}$, $T_4 = \mathtt{banana\$}$, $T_5 = \mathtt{aba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

$$\mathsf{r}_{1,2} = 0^1 1^1 0^1 \underline{1^2} 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 \qquad \mathsf{r}_{1,4} = 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1$$

$$\mathsf{r}_{1,3} = 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 \underline{0^2 1^0} \qquad \mathsf{r}_{1,5} = 0^1 1^1 0^1 \underline{1^2} \underline{0^2} 1^1 \underline{0^2 1^0}$$

Algorithm 1

Steps:

1. Compute BWT and DA for $T^{cat} = T_1 T_2 \ldots T_d$.
2. Build d bitvectors $\mathsf{B}_i[1, N]$, where $\mathsf{B}_i[j] = 1$ if $\mathsf{DA}[j] = i$ with rank/select support.
3. For each string T_i, compute $\mathsf{r_{i,j}}$, with $j > i$:
   - 3.1 Select the intervals $\mathsf{DA}[a, b]$ that contain consecutive entries of i.
   - 3.2 Count k_j occurrences of $j \Rightarrow 0^1 1^{k_j}$, $\mathsf{rank}_1(\mathsf{B}_j, b) - \mathsf{rank}_1(\mathsf{B}_j, a)$.
   - 3.3 Whenever $j \notin \mathsf{DA}[a, b]$, we collapse $0^{\ell_j} 1^0 0^1 \Rightarrow 0^{\ell_j + 1}$.

$T_1 = \mathtt{banana\$}$, $T_2 = \mathtt{anaba\$}$, $T_3 = \mathtt{ba\$}$, $T_4 = \mathtt{banana\$}$, $T_5 = \mathtt{aba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

$$\mathsf{r}_{1,2} = 0^1 1^1 0^1 \underline{1^2} 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 \qquad \mathsf{r}_{1,4} = 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^1$$

$$\mathsf{r}_{1,3} = 0^1 1^1 0^1 1^1 \underline{0^2} 1^1 \underline{0^3} \qquad \mathsf{r}_{1,5} = 0^1 1^1 0^1 \underline{1^2} \underline{0^2} 1^1 \underline{0^3}$$

Algorithm 1

Running time:

- For each string T_i: n_i select and $\approx n_i \times d$ rank operations $\leftarrow O(n_i \times d)$-time
- $O(dN)$-time to compute $M_{d \times d}$ $\leftarrow$ compute one BWT $O(N)$-time

Working space:

- $dN + o(dN)$ bits for the bitvectors with rank/select support. (DA is replaced by $B_1, B_2, \ldots, B_d$)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4
B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Alternatives:

1. Sparse bitvectors: $\uparrow O(dN \times \log \frac{N}{avg(n_i)})$-time.
2. Wavelet trees: $\uparrow O(dN \times \lg d)$-time.

Each rank/select query in $O(1)$ time.

Algorithm 1

Running time:

- For each string T_i: n_i select and $\approx n_i \times d$ rank operations $\leftarrow O(n_i \times d)$-time
- $O(dN)$-time to compute $\mathrm{M}_{d \times d}$ $\leftarrow$ compute one BWT $O(N)$-time

Working space:

- $dN + o(dN)$ bits for the bitvectors with rank/select support. (DA is replaced by $\mathrm{B}_1, \mathrm{B}_2, \ldots, \mathrm{B}_d$)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4
B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Alternatives:

1. Sparse bitvectors: $\uparrow O(dN \times \log \frac{N}{avg(n_i)})$-time.
2. Wavelet trees: $\uparrow O(dN \times \lg d)$-time.

.

Algorithm 1

Running time:

- For each string T_i: n_i select and $\approx n_i \times d$ rank operations $\leftarrow O(n_i \times d)$-time
- $O(dN)$-time to compute $\mathrm{M}_{d \times d}$ $\leftarrow$ compute one BWT $O(N)$-time

Working space:

- $dN + o(dN)$ bits for the bitvectors with rank/select support. (DA is replaced by $\mathrm{B}_1, \mathrm{B}_2, \ldots, \mathrm{B}_d$)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Alternatives:

1. Sparse bitvectors: $\uparrow O(dN \times \log \frac{N}{avg(n_i)})$-time.
2. Wavelet trees: $\uparrow O(dN \times \lg d)$-time.

.

Algorithm 1

Running time:

- For each string T_i: n_i select and $\approx n_i \times d$ rank operations $\leftarrow O(n_i \times d)$-time
- $O(dN)$-time to compute $\mathsf{M}_{d \times d}$ $\leftarrow$ compute one BWT $O(N)$-time

Working space:

- $dN + o(dN)$ bits for the bitvectors with rank/select support. (DA is replaced by $\mathsf{B}_1, \mathsf{B}_2, \ldots, \mathsf{B}_d$)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4
B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Alternatives:

1. Sparse bitvectors: $\uparrow O(dN \times \log \frac{N}{avg(n_i)})$-time.
2. Wavelet trees: $\uparrow O(dN \times \lg d)$-time.

We evaluate these alternatives in practice.

Algorithm 1

Running time:

- For each string T_i: n_i select and $\approx n_i \times d$ rank operations $\leftarrow O(n_i \times d)$-time
- $O(dN)$-time to compute $\mathrm{M}_{d \times d}$ $\leftarrow$ compute one BWT $O(N)$-time

Working space:

- $dN + o(dN)$ bits for the bitvectors with rank/select support. (DA is replaced by $\mathrm{B}_1, \mathrm{B}_2, \ldots, \mathrm{B}_d$)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4
B_1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0
B_2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
B_3	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
B_4	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1
B_5	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Alternatives:

1. Sparse bitvectors: $\uparrow O(dN \times \log \frac{N}{avg(n_i)})$-time.
2. Wavelet trees: $\uparrow O(dN \times \lg d)$-time.

We evaluate these alternatives in practice.

Outline

1. Introduction

2. Burrows-Wheeler Similarity Distribution (BWSD)

3. Algorithm 1

4. Algorithm 2

5. Experiments

6. References

Algorithm 2

Steps:

- Compute BWT and DA for $T^{cat} = T_1 T_2 \ldots T_d$.
- For $i = 1, 2, \ldots, N$ do
  - Solve *document-listing* problem for DA$[i, next[i]]$;
  - All r distinct documents with frequencies in the interval $\leftarrow O(r)$-time.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

Algorithm 2

Steps:

- Compute BWT and DA for $T^{cat} = T_1 T_2 \dots T_d$.
- For $i = 1, 2, \dots, N$ do
  - Solve *document-listing* problem for $\mathrm{DA}[i, \mathrm{next}[i]]$:
  - All r distinct documents with frequencies in the interval $\leftarrow$ $O(r)$-time.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

Algorithm 2

Steps:

- Compute BWT and DA for $T^{cat} = T_1 T_2 \ldots T_d$.
- For $i = 1, 2, \ldots, N$ do
  - Solve *document-listing* problem for $\mathrm{DA}[i, \mathrm{next}[i]]$:
  - All r distinct documents with frequencies in the interval $\leftarrow$ $O(r)$-time.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

$\mathrm{r}_{1,2} = 0^1 1^1 0^1$ $\quad$ $\mathrm{r}_{2,3} = 0^1$ $\quad$ $\mathrm{r}_{3,4} = 0^1 1^1$ $\quad$ $\mathrm{r}_{4,5} =$

$\mathrm{r}_{1,3} = 0^1 1^1 0^1$ $\quad$ $\mathrm{r}_{2,4} =$ $\quad$ $\mathrm{r}_{3,5} = 0^1 1^1$

$\mathrm{r}_{1,4} = 0^1 1^1$ $\quad$ $\mathrm{r}_{2,5} =$

$\mathrm{r}_{1,5} = 0^1 1^1$

$R[i] = rank_{\mathrm{DA}[i]}(\mathrm{DA}, i)$, allows to get the frequencies in $O(1)$ time.

Algorithm 2

Steps:

- Compute BWT and DA for $T^{cat} = T_1 T_2 \dots T_d$.
- For $i = 1, 2, \dots, N$ do
  - Solve *document-listing* problem for $\mathrm{DA}[i, \mathrm{next}[i]]$:
  - All r distinct documents with frequencies in the interval $\leftarrow$ $O(r)$-time.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

$\mathrm{r}_{1,2} = 0^1 1^1 0^1$ $\mathrm{r}_{2,3} = 0^1 1^1 0^1$ $\mathrm{r}_{3,4} = 0^1 1^1$ $\mathrm{r}_{4,5} =$

$\mathrm{r}_{1,3} = 0^1 1^1 0^1$ $\mathrm{r}_{2,4} = 0^1 1^1$ $\mathrm{r}_{3,5} = 0^1 1^1$

$\mathrm{r}_{1,4} = 0^1 1^1$ $\mathrm{r}_{2,5} = 0^1 1^1$

$\mathrm{r}_{1,5} = 0^1 1^1$

$R[i] = rank_{\mathrm{DA}[i]}(\mathrm{DA}, i)$, allows to get the frequencies in $O(1)$ time.

Algorithm 2

Steps:

- Compute BWT and DA for $T^{cat} = T_1 T_2 \ldots T_d$.
- For $i = 1, 2, \ldots, N$ do
  - Solve *document-listing* problem for $\underline{\mathrm{DA}[i, \mathrm{next}[i]]}$:
  - All r distinct documents with frequencies in the interval $\leftarrow$ $\underline{O(r)\text{-time}}$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4
		↑					↑																					

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4
			↑					↑																				

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4
				↑					↑																			

$r_{1,2} = 0^1 1^1 \underline{0^1}$ $\quad r_{2,3} = 0^1 1^1 \underline{0^1}$ $\quad r_{3,4} = 0^1 1^1$ $\quad r_{4,5} =$

$r_{1,3} = 0^1 1^1 \underline{0^1}$ $\quad r_{2,4} = 0^1 1^1$ $\quad r_{3,5} = 0^1 1^1$

$r_{1,4} = 0^1 1^1$ $\quad r_{2,5} = 0^1 1^1$

$r_{1,5} = 0^1 1^1$

$R[i] = rank_{\mathrm{DA}[i]}(\mathrm{DA}, i)$, allows to get the frequencies in $O(1)$ time.

Algorithm 2

Running time:

- We compute DA, prev, next, rmq_{prev}, RMQ_{next} and $R \leftarrow O(N)$-time.
- Total $O(N+z)$-time, where z is the sum of all runs in all $r_{i,j}$

Working space:

- Quadratic matrix to store all lists $r_{i,j}$ in memory (counters $0^1 1^1 \Rightarrow p^1_{i,j} = 2$).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

$r_{1,2} = 0^1 1^1 0^1$ $\quad r_{2,3} = 0^1 1^1 0^1$ $\quad r_{3,4} = 0^1 1^1$ $\quad r_{4,5} =$

$r_{1,3} = 0^1 1^1 0^1$ $\quad r_{2,4} = 0^1 1^1$ $\quad r_{3,5} = 0^1 1^1$

$r_{1,4} = 0^1 1^1$ $\quad r_{2,5} = 0^1 1^1$

$r_{1,5} = 0^1 1^1$

Alternative:

1. Scan DA[1] ... DA[N] d times, one for each T_i. $\rightarrow$ store only d lists $r_{\text{DA}[i],j}$
2. Running time $\uparrow$ increases to $O(dN)$.

Algorithm 2

Running time:

- We compute DA, prev, next, rmq_{prev}, RMQ_{next} and $R \leftarrow O(N)$-time.
- Total $O(N+z)$-time, where z is the sum of all runs in all $r_{i,j}$

Working space:

- Quadratic matrix to store all lists $r_{i,j}$ in memory (counters $0^1 1^1 \Rightarrow p^1_{i,j} = 2$).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4
				↑					↑																			

$r_{1,2} = 0^1 1^1 0^1$ $\quad r_{2,3} = 0^1 1^1 0^1$ $\quad r_{3,4} = 0^1 1^1$ $\quad r_{4,5} =$

$r_{1,3} = 0^1 1^1 0^1$ $\quad r_{2,4} = 0^1 1^1$ $\quad r_{3,5} = 0^1 1^1$

$r_{1,4} = 0^1 1^1$ $\quad r_{2,5} = 0^1 1^1$

$r_{1,5} = 0^1 1^1$

Alternative:

1. Scan DA[1] ... DA[N] d times, one for each T_i. $\rightarrow$ store only d lists $r_{\text{DA[i]},j}$
2. Running time $\uparrow$ increases to $O(dN)$.

Algorithm 2

Running time:

- We compute DA, prev, next, rmq_{prev}, RMQ_{next} and $R \leftarrow O(N)$-time.
- Total $O(N+z)$-time, where z is the sum of all runs in all $r_{i,j}$

Working space:

- Quadratic matrix to store all lists $r_{i,j}$ in memory (counters $0^1 1^1 \Rightarrow p^1_{i,j} = 2$).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

$r_{1,2} = 0^1 1^1 0^1$ $\quad r_{2,3} = 0^1 1^1 0^1$ $\quad r_{3,4} = 0^1 1^1$ $\quad r_{4,5} =$

$r_{1,3} = 0^1 1^1 0^1$ $\quad r_{2,4} = 0^1 1^1$ $\quad r_{3,5} = 0^1 1^1$

$r_{1,4} = 0^1 1^1$ $\quad r_{2,5} = 0^1 1^1$

$r_{1,5} = 0^1 1^1$

Alternative:

1. Scan DA[1] ... DA[N] d times, one for each T_i. $\rightarrow$ store only d lists $r_{\text{DA}[i],j}$
2. Running time $\uparrow$ increases to $O(dN)$.

Algorithm 2

Running time:

- We compute DA, prev, next, rmq_{prev}, RMQ_{next} and $R \leftarrow O(N)$-time.
- Total $O(N+z)$-time, where z is the sum of all runs in all $r_{i,j}$

Working space:

- Quadratic matrix to store all lists $r_{i,j}$ in memory (counters $0^1 1^1 \Rightarrow p^1_{i,j} = 2$).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4
				↑					↑																			

$r_{1,2} = 0^1 1^1 \underline{0^1}$ $\quad r_{2,3} = 0^1 1^1 \underline{0^1}$ $\quad r_{3,4} = \underline{0^1 1^1}$ $\quad r_{4,5} =$

$r_{1,3} = 0^1 1^1 \underline{0^1}$ $\quad r_{2,4} = 0^1 1^1$ $\quad r_{3,5} = \underline{0^1 1^1}$

$r_{1,4} = 0^1 1^1$ $\quad r_{2,5} = 0^1 1^1$

$r_{1,5} = 0^1 1^1$

Alternative:

1. Scan DA[1] … DA[N] d times, one for each T_i. $\rightarrow$ store only d lists $r_{\text{DA}[i],j}$
2. Running time ↑ increases to $O(dN)$.

Algorithm 2

Running time:

- We compute DA, prev, next, rmq_{prev}, RMQ_{next} and $R \leftarrow O(N)$-time.
- Total $O(N+z)$-time, where z is the sum of all runs in all $\text{r}_{i,j}$

Working space:

- Quadratic matrix to store all lists $\text{r}_{i,j}$ in memory (counters $0^1 1^1 \Rightarrow p^1_{i,j} = 2$).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4
				↑					↑																			

$\text{r}_{1,2} = 0^1 1^1 \underline{0^1}$ $\quad \text{r}_{2,3} = 0^1 1^1 \underline{0^1}$ $\quad \text{r}_{3,4} = \underline{0^1 1^1}$ $\quad \text{r}_{4,5} =$

$\text{r}_{1,3} = 0^1 1^1 \underline{0^1}$ $\quad \text{r}_{2,4} = 0^1 1^1$ $\quad \text{r}_{3,5} = \underline{0^1 1^1}$

$\text{r}_{1,4} = 0^1 1^1$ $\quad \text{r}_{2,5} = 0^1 1^1$

$\text{r}_{1,5} = 0^1 1^1$

Alternative:

1. Scan $\text{DA}[1] \dots \text{DA}[N]$ d times, one for each T_i. $\rightarrow$ store only d lists $\text{r}_{\text{DA}[i],j}$
2. Running time ↑ increases to $O(dN)$.

Outline

1. Introduction

2. Burrows-Wheeler Similarity Distribution (BWSD)

3. Algorithm 1

4. Algorithm 2

5. Experiments

6. References

Experiments

Implementation:

- C++ using SDSL-lite v.2.
- Source code: https://github.com/felipelouza/bwsd.

Algorithms:

- SF: straightforward $\underline{\mathrm{BWT}(T_1, T_2)}, \underline{\mathrm{BWT}(T_1, T_3)}, \ldots, \underline{\mathrm{BWT}(T_{d-1}, T_d)} \leftarrow O(dN)$-time
- Algorithm 1:
  1. BIT: $\underline{d \times \text{bitvectors}}$. $\leftarrow O(dN)$-time
  2. BIT_sd: $\underline{d \times \text{compressed bitvectors}}$ (Elias-Fano). $\leftarrow O(dN \times \log \frac{N}{avg(n_i)})$-time
  3. WT: $\underline{\text{1 wavelet tree}}$. $\leftarrow O(dN \lg d)$-time
- Algorithm 2:
  1. RMQ_opt: $\underline{O(N+z)\text{-time}}$.
  2. RMQ_light: $\underline{\textit{lightweight}}$ version $\leftarrow O(dN)$-time.

Configuration:

- 64 bits Debian GNU/Linux 8 (kernel 3.16.0-4)
- Intel Xeon Processor E5-2630 v3 20M Cache 2.40-GH, 384 GB of RAM.

Experiments

Implementation:

- C++ using SDSL-lite v.2.
- Source code: https://github.com/felipelouza/bwsd.

Algorithms:

- SF: straightforward BWT(T_1, T_2), BWT(T_1, T_3), $\ldots$, BWT(T_{d-1}, T_d) $\leftarrow O(dN)$-time
- Algorithm 1:
  1. BIT: $d \times$ bitvectors. $\leftarrow O(dN)$-time
  2. BIT_sd: $d \times$ compressed bitvectors (Elias-Fano). $\leftarrow O(dN \times \log \frac{N}{avg(n_i)})$-time
  3. WT: 1 wavelet tree. $\leftarrow O(dN \lg d)$-time
- Algorithm 2:
  1. RMQ_opt: $O(N + z)$-time.
  2. RMQ_light: *lightweight* version $\leftarrow O(dN)$-time.

Configuration:

- 64 bits Debian GNU/Linux 8 (kernel 3.16.0-4)
- Intel Xeon Processor E5-2630 v3 20M Cache 2.40-GH, 384 GB of RAM.

Experiments

Implementation:

- C++ using SDSL-lite v.2.
- Source code: https://github.com/felipelouza/bwsd.

Algorithms:

- SF: straightforward $\underline{\mathrm{BWT}(T_1, T_2)}, \underline{\mathrm{BWT}(T_1, T_3)}, \ldots, \underline{\mathrm{BWT}(T_{d-1}, T_d)} \leftarrow O(dN)$-time
- Algorithm 1:
  1. BIT: $\underline{d \times \text{bitvectors}}$. $\leftarrow O(dN)$-time
  2. BIT_sd: $\underline{d \times \text{compressed bitvectors}}$ (Elias-Fano). $\leftarrow O(dN \times \log \frac{N}{avg(n_i)})$-time
  3. WT: $\underline{\text{1 wavelet tree}}$. $\leftarrow O(dN \lg d)$-time
- Algorithm 2:
  1. RMQ_opt: $\underline{O(N+z)\text{-time}}$.
  2. RMQ_light: $\underline{\textit{lightweight}}$ version $\leftarrow O(dN)$-time.

Configuration:

- 64 bits Debian GNU/Linux 8 (kernel 3.16.0-4)
- Intel Xeon Processor E5-2630 v3 20M Cache 2.40-GH, 384 GB of RAM.

Experiments

Implementation:

- C++ using SDSL-lite v.2.
- Source code: https://github.com/felipelouza/bwsd.

Algorithms:

- SF: straightforward $\text{BWT}(T_1, T_2), \text{BWT}(T_1, T_3), \ldots, \text{BWT}(T_{d-1}, T_d) \leftarrow O(dN)$-time
- Algorithm 1:
  1. BIT: $d \times$ bitvectors. $\leftarrow O(dN)$-time
  2. BIT_sd: $d \times$ compressed bitvectors (Elias-Fano). $\leftarrow O(dN \times \log \frac{N}{avg(n_i)})$-time
  3. WT: 1 wavelet tree. $\leftarrow O(dN \lg d)$-time
- Algorithm 2:
  1. RMQ_opt: $O(N + z)$-time.
  2. RMQ_light: *lightweight* version $\leftarrow O(dN)$-time.

Configuration:

- 64 bits Debian GNU/Linux 8 (kernel 3.16.0-4)
- Intel Xeon Processor E5-2630 v3 20M Cache 2.40-GH, 384 GB of RAM.

Experiments

Implementation:

- C++ using SDSL-lite v.2.
- Source code: https://github.com/felipelouza/bwsd.

Algorithms:

- SF: straightforward $\underline{\mathrm{BWT}(T_1, T_2)}, \underline{\mathrm{BWT}(T_1, T_3)}, \ldots, \underline{\mathrm{BWT}(T_{d-1}, T_d)} \leftarrow O(dN)$-time
- Algorithm 1:
  1. `BIT`: $\underline{d \times \text{bitvectors}}$. $\leftarrow O(dN)$-time
  2. `BIT_sd`: $\underline{d \times \text{compressed bitvectors}}$ (Elias-Fano). $\leftarrow O(dN \times \log \frac{N}{avg(n_i)})$-time
  3. `WT`: $\underline{1 \text{ wavelet tree}}$. $\leftarrow O(dN \lg d)$-time
- Algorithm 2:
  1. `RMQ_opt`: $\underline{O(N+z)\text{-time}}$.
  2. `RMQ_light`: *lightweight* version $\leftarrow O(dN)$-time.

Configuration:

- 64 bits Debian GNU/Linux 8 (kernel 3.16.0-4)
- Intel Xeon Processor E5-2630 v3 20M Cache 2.40-GH, 384 GB of RAM.

Experiments

Datasets:

- We used $d = 15.000$ strings from datasets:

dataset	σ	total length	n. of strings	max length	avg length
READS	4	1,422,718	15,000	101	94.85
UNIPROT	25	3,454,210	15,000	2,147	230.28
ESTS	4	11,313,165	15,000	1,560	754.21
WIKIPEDIA	208	25,430,657	15,000	150,768	1,695.38

READS: collection of reads from Human Chromosome 14 (library 1).

UNIPROT: collection of protein sequences from Uniprot/TrEMBL protein database release 2015_09.

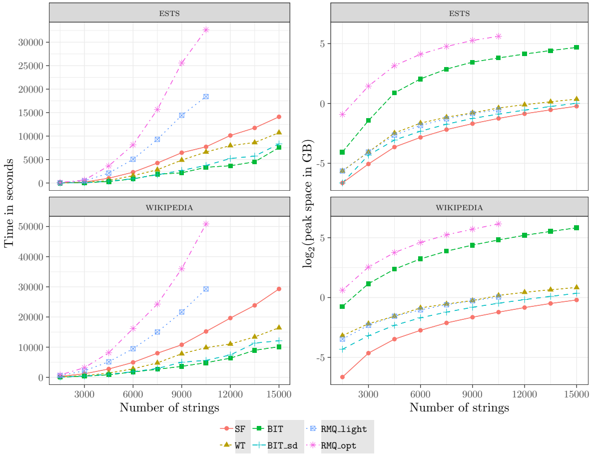
ESTS: collection of DNA sequences of ESTs from *C. elegans*.

WIKIPEDIA: collection of pages from a snapshot of the English-language edition of Wikipedia.

Experiments

Running time[3] and Peak space:

- Alg. 1 was the fastest: `BIT` was 2.4 × faster, and `BIT_sd` 2.0 × faster than `SF`.
- All versions of Alg. 2 were the slowest: we stopped with $d = 10,500$ strings.

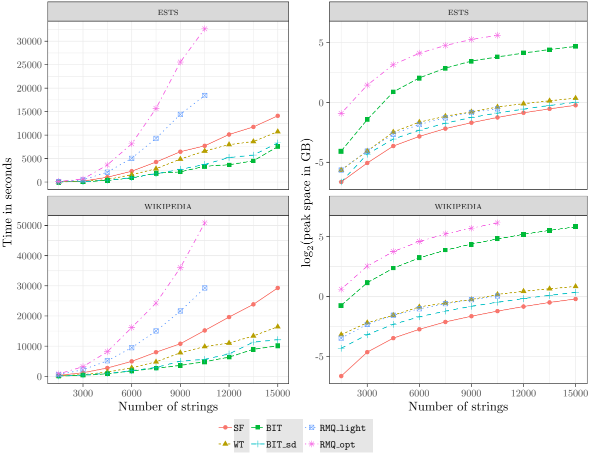


[3] Time to build auxiliary data structures: less than 1% of the total.

Experiments

Running time[3] and Peak space:

- Alg. 1 was the fastest: BIT was 2.4 × faster, and BIT_sd 2.0 × faster than SF.
- All versions of Alg. 2 were the slowest: we stopped with $d = 10,500$ strings.

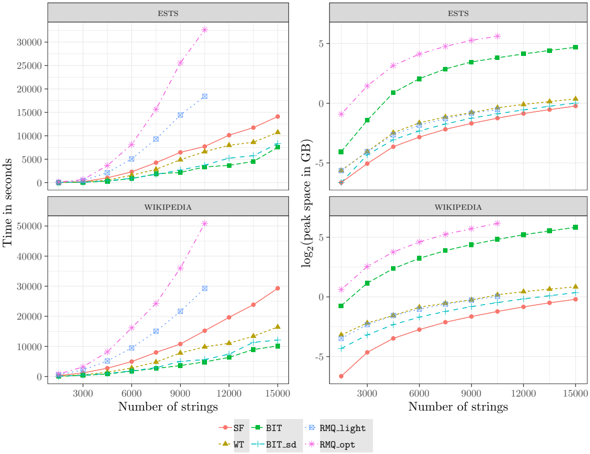


[3] Time to build auxiliary data structures: less than 1% of the total.

Experiments

Running time and Peak space:

- The space used by BIT was very large: BIT used 64 × more space than SF[4].
- BIT_sd provides good space-efficient alternative: it used 1.5 × the space of SF.

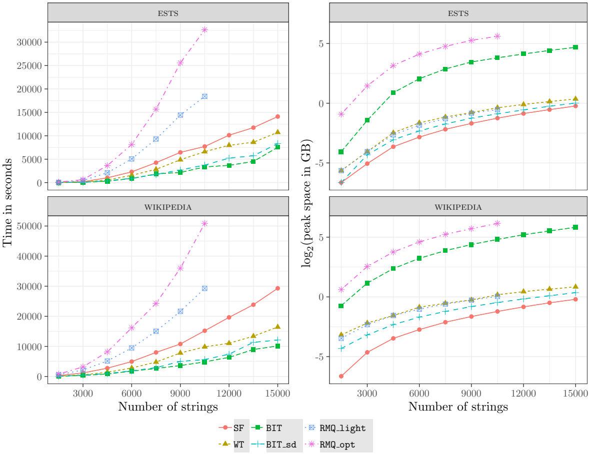


[4] SF: peakspace very close to I/O size: input collection and output matrix.

Experiments

Running time and Peak space:

- The space used by BIT was very large: BIT used 64 × more space than SF[4].
- BIT_sd provides good space-efficient alternative: it used 1.5 × the space of SF.

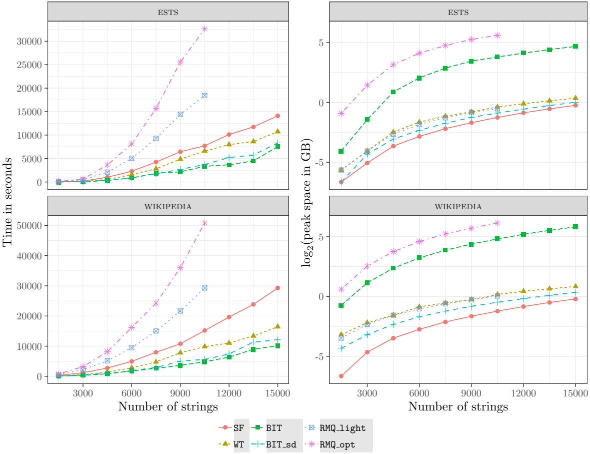


[4] SF: peakspace very close to I/O size: input collection and output matrix.

Experiments

Running time and Peak space:

- The space used by BIT was very large: BIT used 64 × more space than SF[4].
- BIT_sd provides good space-efficient alternative: it used 1.5 × the space of SF.



[4]SF: peakspace very close to I/O size: input collection and output matrix.

Experiments

Artificial input:

- All strings completely "different" $\Rightarrow$ DA $= 1^{N/d}2^{N/d}\ldots d^{N/d}$ (composed by d runs).
- In the extreme case, Alg. 2 (`RMQ_opt`) runs in $O(N)$-time.

Running time:

- Alg. 1 is still the fastest,
- `RMQ_opt` and `RMQ_light`: very close.

Example: strings from interleaved and disjoint alphabets.

Experiments

Artificial input:

- All strings completely "different" $\Rightarrow$ DA $= 1^{N/d} 2^{N/d} \ldots d^{N/d}$ (composed by d runs).
- In the extreme case, Alg. 2 (RMQ_opt) runs in $O(N)$-time.

Running time:

- Alg. 1 is still the fastest, Alg. 2 was 2.75 × faster than SF.
- RMQ_opt and RMQ_light: very close.

$O(N + d)$-time versus $O(Nd)$-time (others).

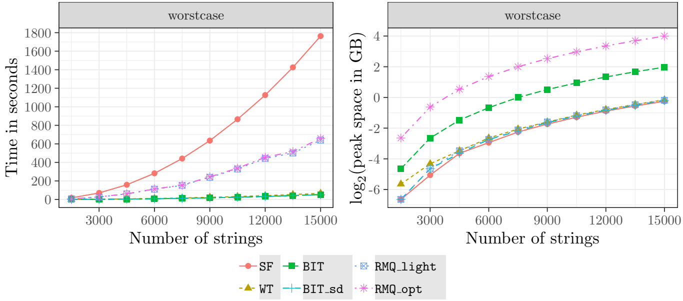
Experiments

Artificial input:

- All strings completely "different" $\Rightarrow$ DA $= 1^{N/d}2^{N/d}\ldots d^{N/d}$ (composed by d runs).
- In the extreme case, Alg. 2 (RMQ_opt) runs in $O(N)$-time.

Running time:

- Alg. 1 is still the fastest, Alg. 2 was 2.75 $\times$ faster than SF.
- RMQ_opt and RMQ_light: very close.



Alg. 1 implementations avoid rank queries (next is not in the interval).

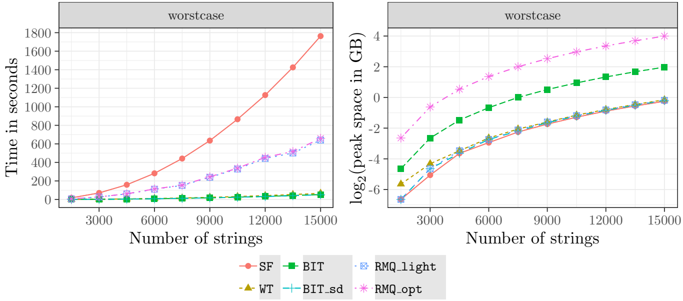
Experiments

Artificial input:

- All strings completely "different" $\Rightarrow$ DA $= 1^{N/d}2^{N/d}\ldots d^{N/d}$ (composed by d runs).
- In the extreme case, Alg. 2 (RMQ_opt) runs in $O(N)$-time.

Running time:

- Alg. 1 is still the fastest, Alg. 2 was 2.75 $\times$ faster than SF.
- RMQ_opt and RMQ_light: very close.



Alg. 1 implementations avoid rank queries (next is not in the interval).

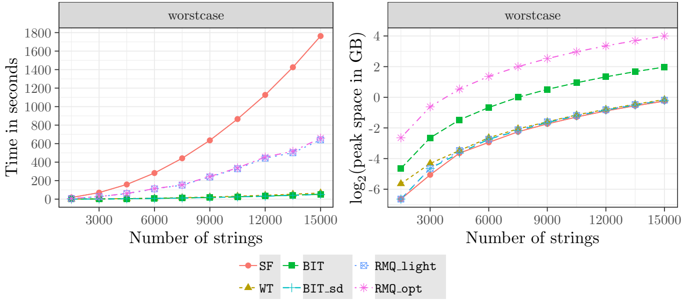
Experiments

Artificial input:

- All strings completely "different" $\Rightarrow$ DA $= 1^{N/d} 2^{N/d} \ldots d^{N/d}$ (composed by d runs).
- In the extreme case, Alg. 2 (RMQ_opt) runs in $O(N)$-time.

Running time:

- Alg. 1 is still the fastest, Alg. 2 was 2.75 × faster than SF.
- RMQ_opt and RMQ_light: very close.



$O(N + d)$-time versus $O(Nd)$-time (others).

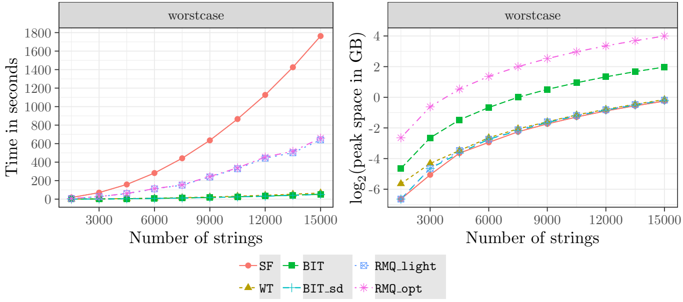
Experiments

Artificial input:

- All strings completely "different" $\Rightarrow \text{DA} = 1^{N/d}2^{N/d}\ldots d^{N/d}$ (composed by d runs).
- In the extreme case, Alg. 2 (RMQ_opt) runs in $O(N)$-time.

Running time:

- Alg. 1 is still the fastest, Alg. 2 was 2.75 × faster than SF.
- RMQ_opt and RMQ_light: very close.



Peak space was the same (previous results).

Muchas gracias!

louza@usp.br

Outline

1. Introduction

2. Burrows-Wheeler Similarity Distribution (BWSD)

3. Algorithm 1

4. Algorithm 2

5. Experiments

6. References

Felipe A. Louza, Simon Gog, and Guilherme P. Telles.
Inducing enhanced suffix arrays for string collections.
Theor. Comput. Sci., 678:22–39, 2017.

S. Mantaci, a. Restivo, G. Rosone, and M. Sciortino.
A new combinatorial approach to sequence comparison.
Theory of Computing Systems, 42(3):411–429, 2008.

Lianping Yang, Guisong Chang, Xiangde Zhang, and Tianming Wang.
Use of the Burrows-Wheeler similarity distribution to the comparison of the proteins.
Amino Acids, 39(3):887–898, 2010.

Lianping Yang, Xiangde Zhang, and Tianming Wang.
The Burrows-Wheeler similarity distribution between biological sequences based on Burrows-Wheeler transform.
Journal of Theoretical Biology, 262(4):742–749, 2010.

Introduction

The terminator problem:

- Using distinct terminators $\$_i$ $\Rightarrow$ increases the alphabet size to $\sigma + d$.

$$T^{cat} = \underline{T_1[1, n_1 - 1]\$_1} \cdot \underline{T_2[1, n_2 - 1]\$_2} \cdots \underline{T_d[1, n_d - 1]\$_d}$$

- SA may be build using only one additional terminator # by gSACA-K [LGT17]:

$$T^{cat} = \underline{T_1[1, n_1 - 1]\$} \cdot \underline{T_2[1, n_2 - 1]\$} \cdots \underline{T_d[1, n_d - 1]\$}\#$$

- SA is built in $O(N)$ time, such that \$-symbols are ordered:

$$T^{cat}[i] = \$ < T^{cat}[j] = \$ \text{ iff } i < j$$

Introduction

The terminator problem:

- Using distinct terminators $\$_i$ $\Rightarrow$ increases the alphabet size to $\sigma + d$.

$$T^{cat} = \underline{T_1[1, n_1 - 1]\$_1} \cdot \underline{T_2[1, n_2 - 1]\$_2} \cdots \underline{T_d[1, n_d - 1]\$_d}$$

- SA may be build using only one additional terminator $\#$ by gSACA-K [LGT17]:

$$T^{cat} = \underline{T_1[1, n_1 - 1]\$} \cdot \underline{T_2[1, n_2 - 1]\$} \cdots \underline{T_d[1, n_d - 1]\$}\#$$

- SA is built in $O(N)$ time, such that \$-symbols are ordered:

$$T^{cat}[i] = \$ < T^{cat}[j] = \$ \text{ iff } i < j$$

Introduction

The terminator problem:

- Using distinct terminators $\$_i$ $\Rightarrow$ increases the alphabet size to $\sigma + d$.

$$T^{cat} = \underline{T_1[1, n_1 - 1]\$_1} \cdot \underline{T_2[1, n_2 - 1]\$_2} \cdots \underline{T_d[1, n_d - 1]\$_d}$$

- SA may be build using only one additional terminator $\#$ by gSACA-K [LGT17]:

$$T^{cat} = \underline{T_1[1, n_1 - 1]\$} \cdot \underline{T_2[1, n_2 - 1]\$} \cdots \underline{T_d[1, n_d - 1]\$}\#$$

- SA is built in $O(N)$ time, such that \$-symbols are ordered:

$$T^{cat}[i] = \$ < T^{cat}[j] = \$ \text{ iff } i < j$$

Equivalent to $\$_1 < \$_2 < \cdots < \$_d$;

Algorithm 2

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

. . .

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4

. . .

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BWT	$	a	a	a	a	a	n	b	b	n	b	n	$	n	n	$	b	b	a	$	a	#	$	a	a	a	a	a
DA	6	1	2	3	4	5	1	2	3	4	5	2	5	1	4	2	1	4	2	3	5	1	4	1	4	2	1	4