Efficient Construction of the BWT for Repetitive Text Using String Compression

Diego Díaz[1] Gonzalo Navarro[2]

[1]Department of Computer Science, University of Helsinki

[2]Department of Computer Science, University of Chile

CPM 2022, Prague

June 27, 2022

Motivation

The *Burrows–Wheeler Transform* (BWT) is an important string transformation used for compressing and indexing text.

Motivation

The *Burrows–Wheeler Transform* (BWT) is an important string transformation used for compressing and indexing text.

There are popular linear-time and linear-space algorithms for constructing the BWT of a text.

Motivation

The *Burrows–Wheeler Transform* (BWT) is an important string transformation used for compressing and indexing text.

There are popular linear-time and linear-space algorithms for constructing the BWT of a text.

Still ...

Motivation

The *Burrows–Wheeler Transform* (BWT) is an important string transformation used for compressing and indexing text.

There are popular linear-time and linear-space algorithms for constructing the BWT of a text.

These algorithms are still impractical for applications where the input text is massive (e.g., Genomics).

Motivation

The *Burrows–Wheeler Transform* (BWT) is an important string transformation used for compressing and indexing text.

There are popular linear-time and linear-space algorithms for constructing the BWT of a text.

These algorithms are still impractical for applications where the input text is massive (e.g., Genomics).

Possible solution:

Motivation

The *Burrows–Wheeler Transform* (BWT) is an important string transformation used for compressing and indexing text.

There are popular linear-time and linear-space algorithms for constructing the BWT of a text.

These algorithms are still impractical for applications where the input text is massive (e.g., Genomics).

Possible solution:

We require BWT algorithms with a cost proportional to the amount of information in the input, not the input size.

Motivation

The *Burrows–Wheeler Transform* (BWT) is an important string transformation used for compressing and indexing text.

There are popular linear-time and linear-space algorithms for constructing the BWT of a text.

These algorithms are still impractical for applications where the input text is massive (e.g., Genomics).

Possible solution:

We require BWT algorithms with a cost proportional to the amount of information in the input, not the input size.

We refer to this type of methods as repetition-aware.

Our contribution

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of k strings and $n = \Sigma_1^k |T_i|$ symbols.

Our contribution

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of k strings and $n = \Sigma_1^k |T_i|$ symbols.

Our contribution

We present a repetition-aware and semi-external algorithm for constructing the BCR BWT of $\mathcal{T}$ that runs in $O(n)$ time and uses $O(n)$ bits of working memory.

Our contribution

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of k strings and $n = \Sigma_1^k |T_i|$ symbols.

Our contribution

We present a repetition-aware and semi-external algorithm for constructing the BCR BWT of $\mathcal{T}$ that runs in $O(n)$ time and uses $O(n)$ bits of working memory.

Important aspects of our method:

Our contribution

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of k strings and $n = \Sigma_1^k |T_i|$ symbols.

Our contribution

We present a repetition-aware and semi-external algorithm for constructing the BCR BWT of $\mathcal{T}$ that runs in $O(n)$ time and uses $O(n)$ bits of working memory.

Important aspects of our method:

- It relies on induced suffix sorting (ISS).

Our contribution

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of k strings and $n = \Sigma_1^k |T_i|$ symbols.

Our contribution

We present a repetition-aware and semi-external algorithm for constructing the BCR BWT of $\mathcal{T}$ that runs in $O(n)$ time and uses $O(n)$ bits of working memory.

Important aspects of our method:

- It relies on induced suffix sorting (ISS).
- We use run-length and grammar-like compression to maintain temporary data in compact form and operate faster than in a plain setting.

Induced suffix sorting (ISS)

ISS is a method developed by **Ko et al. 2005** and **Nong et al. 2009** to build the suffix array in linear time. **Okanohara et al. 2009** adapted ISS to compute the BWT without producing the suffix array.

Induced suffix sorting (ISS)

ISS is a method developed by **Ko et al. 2005** and **Nong et al. 2009** to build the suffix array in linear time. **Okanohara et al. 2009** adapted ISS to compute the BWT without producing the suffix array.

Relevant definitions:

Induced suffix sorting (ISS)

ISS is a method developed by **Ko et al. 2005** and **Nong et al. 2009** to build the suffix array in linear time. **Okanohara et al. 2009** adapted ISS to compute the BWT without producing the suffix array.

Relevant definitions:

- L-type suffixes (L):

Induced suffix sorting (ISS)

ISS is a method developed by **Ko et al. 2005** and **Nong et al. 2009** to build the suffix array in linear time. **Okanohara et al. 2009** adapted ISS to compute the BWT without producing the suffix array.

Relevant definitions:

- L-type suffixes (L):

(A)	t	c	. . .
	L		

(B)	t	t	t	c	. . .
	L	L	L		

Induced suffix sorting (ISS)

ISS is a method developed by **Ko et al. 2005** and **Nong et al. 2009** to build the suffix array in linear time. **Okanohara et al. 2009** adapted ISS to compute the BWT without producing the suffix array.

Relevant definitions:

- L-type suffixes (L):
- S-type suffixes (S):

Induced suffix sorting (ISS)

ISS is a method developed by **Ko et al. 2005** and **Nong et al. 2009** to build the suffix array in linear time. **Okanohara et al. 2009** adapted ISS to compute the BWT without producing the suffix array.

Relevant definitions:

- L-type suffixes (L):
- S-type suffixes (S):

(A) $\begin{array}{ccc} c & t & \ldots \\ S & & \end{array}$ **(B)** $\begin{array}{ccccc} \texttt{c} & \texttt{c} & \texttt{c} & \texttt{t} & \ldots \\ S & S & S & & \end{array}$

Induced suffix sorting (ISS)

ISS is a method developed by **Ko et al. 2005** and **Nong et al. 2009** to build the suffix array in linear time. **Okanohara et al. 2009** adapted ISS to compute the BWT without producing the suffix array.

Relevant definitions:

- L-type suffixes (L):
- S-type suffixes (S):
- LMS-type suffixes ($S*$):

Induced suffix sorting (ISS)

ISS is a method developed by **Ko et al. 2005** and **Nong et al. 2009** to build the suffix array in linear time. **Okanohara et al. 2009** adapted ISS to compute the BWT without producing the suffix array.

Relevant definitions:

- L-type suffixes (L):
- S-type suffixes (S):
- LMS-type suffixes ($S*$):

c	a	a	...
L	$S*$	S	

Induced suffix sorting (ISS)

ISS is a method developed by **Ko et al. 2005** and **Nong et al. 2009** to build the suffix array in linear time. **Okanohara et al. 2009** adapted ISS to compute the BWT without producing the suffix array.

Relevant definitions:

- L-type suffixes (L):
- S-type suffixes (S):
- LMS-type suffixes ($S*$):
- LMS-type substrings:

Induced suffix sorting (ISS)

ISS is a method developed by **Ko et al. 2005** and **Nong et al. 2009** to build the suffix array in linear time. **Okanohara et al. 2009** adapted ISS to compute the BWT without producing the suffix array.

Relevant definitions:

- L-type suffixes (L):
- S-type suffixes (S):
- LMS-type suffixes ($S*$):
- LMS-type substrings:

t	c	g	g	t	a	g	...
L	$S*$	S	S	L	$S*$	L	

Our method

Our method

Input:

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of k strings and $n = \Sigma_1^k |T_i|$ symbols for which we require to build the BCR BWT.

Our method

Input:
Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of k strings and $n = \Sigma_1^k |T_i|$ symbols for which we require to build the BCR BWT.

Observation

Let $\mathcal{S}$ be the set of distinct strings of length > 1 appearing as suffixes in the LMS substrings of $\mathcal{T}$. $\mathcal{S}$ induces a partition in the suffix array associated with the BCR BWT of $\mathcal{T}$.

Our method

Input:
Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of k strings and $n = \Sigma_1^k |T_i|$ symbols for which we require to build the BCR BWT.

Observation

Let $\mathcal{S}$ be the set of distinct strings of length > 1 appearing as suffixes in the LMS substrings of $\mathcal{T}$. $\mathcal{S}$ induces a partition in the suffix array associated with the BCR BWT of $\mathcal{T}$.

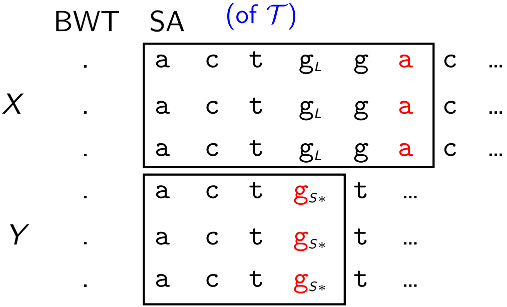
All the suffixes of $\mathcal{T}$ prefixed by some string $Y \in \mathcal{S}$ appear consecutively in the suffix array.

Our method

Consider the strings $X = \texttt{actgga}$ and $Y = \texttt{actg}$. Assume both appear as suffixes in the LMS substrings of $\mathcal{T}$.

Our method

Consider the strings $X = \texttt{actgga}$ and $Y = \texttt{actg}$. Assume both appear as suffixes in the LMS substrings of $\mathcal{T}$.



	BWT	SA		(of $\mathcal{T}$)					
	.	a	c	t	g_L	g	a	c	...
X	.	a	c	t	g_L	g	a	c	...
	.	a	c	t	g_L	g	a	c	...
	.	a	c	t	g_{S*}	t	...		
Y	.	a	c	t	g_{S*}	t	...		
	.	a	c	t	g_{S*}	t	...		

Our observation holds even if Y is prefix of X (or vice-versa)

Our method

Our idea: we use the partition in the SA induced by the LMS substrings of $\mathcal{T}$ to fill as many positions in the BWT as possible.

Our method

Our idea: we use the partition in the SA induced by the LMS substrings of $\mathcal{T}$ to fill as many positions in the BWT as possible.

Let D be the set of strings occurring as LMS substrings in $\mathcal{T}$.

Our method

Our idea: we use the partition in the SA induced by the LMS substrings of $\mathcal{T}$ to fill as many positions in the BWT as possible.

Let D be the set of strings occurring as LMS substrings in $\mathcal{T}$.

Let $Y = \texttt{actg} \in \mathcal{S}$ be a string that appears as a suffix in the strings of D.

Our method

Our idea: we use the partition in the SA induced by the LMS substrings of $\mathcal{T}$ to fill as many positions in the BWT as possible.

Let D be the set of strings occurring as LMS substrings in $\mathcal{T}$.

Let $Y = \mathtt{actg} \in \mathcal{S}$ be a string that appears as a suffix in the strings of D.

We distinguish three cases to fill the BWT range mapping the partition block for Y:

Our method

Our idea: we use the partition in the SA induced by the LMS substrings of $\mathcal{T}$ to fill as many positions in the BWT as possible.

Let D be the set of strings occurring as LMS substrings in $\mathcal{T}$.

Let $Y = \texttt{actg} \in \mathcal{S}$ be a string that appears as a suffix in the strings of D.

We distinguish three cases to fill the BWT range mapping the partition block for Y:

Case 1: if Y is always a proper suffix that is preceded by the same character in D, then the SA block for Y maps an equal-symbol run in the BWT.

BWT	SA					
a	a	c	t	g_{S*}	t	...
a	a	c	t	g_{S*}	t	...
a	a	c	t	g_{S*}	t	...

Our method

Our idea: we use the partition in the SA induced by the LMS substrings of $\mathcal{T}$ to fill as many positions in the BWT as possible.

Let D be the set of strings occurring as LMS substrings in $\mathcal{T}$.

Let $Y = \texttt{actg} \in \mathcal{S}$ be a string that appears as a suffix in the strings of D.

We distinguish three cases to fill the BWT range mapping the partition block for Y:

Case 2: if Y is not a proper suffix in the LMS substrings, then we cannot infer the BWT block for Y using D.

BWT	SA					
*	$\texttt{a}_{S*}$	c	t	$\texttt{g}_{S*}$	t	...
*	$\texttt{a}_{S*}$	c	t	$\texttt{g}_{S*}$	t	...
*	$\texttt{a}_{S*}$	c	t	$\texttt{g}_{S*}$	t	...

Our method

Our idea: we use the partition in the SA induced by the LMS substrings of $\mathcal{T}$ to fill as many positions in the BWT as possible.

Let D be the set of strings occurring as LMS substrings in $\mathcal{T}$.

Let $Y = \texttt{actg} \in \mathcal{S}$ be a string that appears as a suffix in the strings of D.

We distinguish three cases to fill the BWT range mapping the partition block for Y:

Case 3: if Y is not left-maximal, then we cannot infer the BWT block for Y either.

BWT	SA					
a	a	c	t	$\texttt{g}_{S*}$	t	...
$	a	c	t	$\texttt{g}_{S*}$	t	...
*	$\texttt{a}_{S*}$	c	t	$\texttt{g}_{S*}$	t	...

Our method

Our method (like ISS) is recursive.

Our method

Our method (like ISS) is recursive.

In each recursive step i, we proceed as follows:

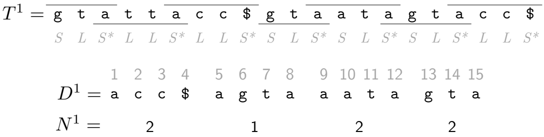
Our method

Entering the recursion:

$$T^1 = \begin{array}{ccccccccccccccccccccc} \overline{\mathtt{g}} & \overline{\mathtt{t}} & \overline{\underline{\mathtt{a}}} & \underline{\mathtt{t}} & \underline{\mathtt{t}} & \underline{\overline{\mathtt{a}}} & \overline{\mathtt{c}} & \overline{\mathtt{c}} & \overline{\mathtt{\$}} & \underline{\mathtt{g}} & \underline{\mathtt{t}} & \underline{\overline{\mathtt{a}}} & \overline{\mathtt{a}} & \overline{\mathtt{t}} & \overline{\underline{\mathtt{a}}} & \underline{\mathtt{g}} & \underline{\mathtt{t}} & \underline{\overline{\mathtt{a}}} & \overline{\mathtt{c}} & \overline{\mathtt{c}} & \overline{\mathtt{\$}} \\ S & L & S^* & L & L & S^* & L & L & S^* & S & L & S^* & S & L & S^* & S & L & S^* & L & L & S^* \end{array}$$

Our method

Entering the recursion:

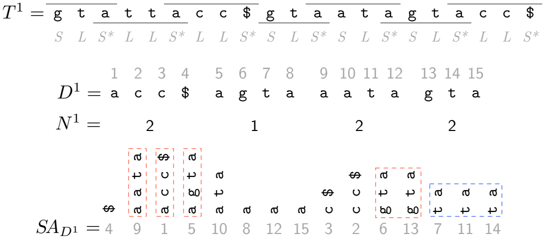


$T^1 =$	g	t	a	t	t	a	c	c	$	g	t	a	a	t	a	g	t	a	c	c	$
	S	L	S^*	L	L	S^*	L	L	S^*	S	L	S^*	S	L	S^*	S	L	S^*	L	L	S^*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$D^1 =$	a	c	c	$	a	g	t	a	a	a	t	a	g	t	a
$N^1 =$		2				1				2				2	

Our method

Entering the recursion:



$T^1 =$ g t a t t a c c $ g t a a t a g t a c c $

S L S* L L S* L L S* S L S* S L S* S L S* L L S*

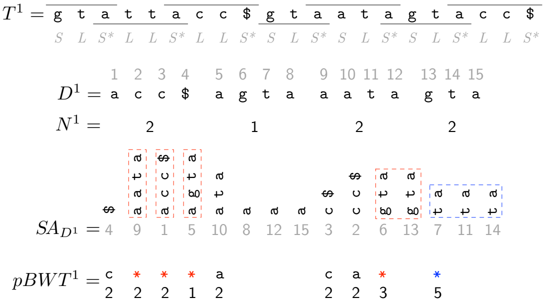
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$D^1 =$	a	c	c	$	a	g	t	a	a	a	t	a	g	t	a

$N^1 =$ 2 1 2 2

$SA_{D^1} =$	4	9	1	5	10	8	12	15	3	2	6	13	7	11	14
	$	aata	acc$	agta	ata	a	a	a	c$	cc$	gta	gta	ta	ta	ta

Our method

Entering the recursion:


$T^1 =$ g t a t t a c c $ g t a a t a g t a c c $

S L $S*$ L L $S*$ L L $S*$ S L $S*$ S L $S*$ S L $S*$ L L $S*$

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

$D^1 =$ a c c $ a g t a a a t a g t a

$N^1 =$ 2 1 2 2

$ aata acc$ agta ata a a a c$ cc$ gta gta ta ta ta

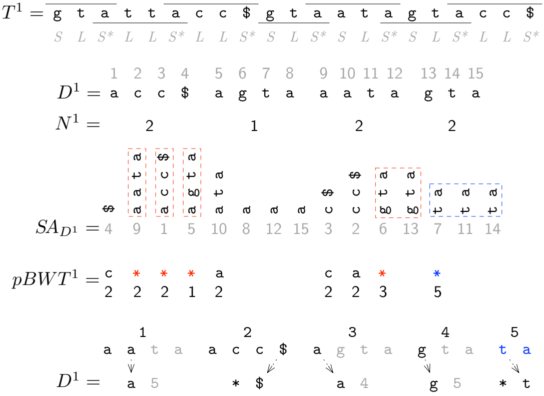
$SA_{D^1} =$ 4 9 1 5 10 8 12 15 3 2 6 13 7 11 14

$pBWT^1 =$ c * * * a c a * *

2 2 2 1 2 2 2 3 5

Our method

Entering the recursion:



$T^1 =$ g t a t t a c c $ g t a a t a g t a c c $

S L $S*$ L L $S*$ L L $S*$ S L $S*$ S L $S*$ S L $S*$ L L $S*$

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

$D^1 =$ a c c $ a g t a a a t a g t a

$N^1 =$ 2 1 2 2

$SA_{D^1} =$ 4 9 1 5 10 8 12 15 3 2 6 13 7 11 14

$pBWT^1 =$ c * * * a c a * *

2 2 2 1 2 2 2 3 5

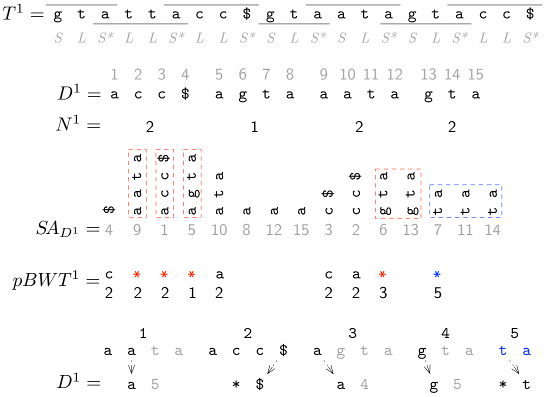
1 2 3 4 5

a a t a a c c $ a g t a g t a t a

$D^1 =$ a 5 * $ a 4 g 5 * t

Our method

Entering the recursion:



$T^1 =$ g t a t t a c c $ g t a a t a g t a c c $

S L $S*$ L L $S*$ L L $S*$ S L $S*$ S L $S*$ S L $S*$ L L $S*$

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

$D^1 =$ a c c $ a g t a a a t a g t a

$N^1 =$ 2 1 2 2

$ | a t a a | $ c c a | a t g a | a t a | a | a | a | $ c | $ c c | a t g | a t g | a t | a t | a t

$SA_{D^1} =$ 4 9 1 5 10 8 12 15 3 2 6 13 7 11 14

$pBWT^1 =$ c * * * a c a * *

2 2 2 1 2 2 2 3 5

1 2 3 4 5

a a t a a c c $ a g t a g t a t a

$D^1 =$ a 5 * $ a 4 g 5 * t

New parse = 4 1 2 4 1 3 2

Our method

Returning from the recursion:

$$T^2 = 4\ 1\ 2\ 4\ 1\ 3\ 2$$

Our method

Returning from the recursion:

$$T^2 = 4\ 1\ 2\ 4\ 1\ 3\ 2$$

BWT^2	
4	1 2
4	1 3 2
1	2
3	2
1	3
2	4 1 2
2	4 1 3 2

Our method

Returning from the recursion:

$$T^2 = 4\ 1\ 2\ 4\ 1\ 3\ 2$$

BWT^2	
4	1 2
4	1 3 2
1	2
3	2
1	3
2	4 1 2
2	4 1 3 2

$PBWT^1$		
	c	2
1	*	2
2	*	2
3	*	1
	a	2
	c	2
	a	2
4	*	3
5	*	5

Our method

Returning from the recursion:

$$T^2 = 4\ 1\ 2\ 4\ 1\ 3\ 2$$

BWT^2		$PBWT^1$		
4	1 2		c	2
4	1 3 2	1	*	2
1	2	2	*	2
3	2	3	*	1
1	3		a	2
2	4 1 2		c	2
2	4 1 3 2		a	2
		4	*	3
		5	*	5

$$D^1 = \overset{1}{\texttt{a}\ 5}\quad \overset{2}{*\ \$}\quad \overset{3}{\texttt{a}\ 4}\quad \overset{4}{\texttt{g}\ 5}\quad \overset{5}{*\ \texttt{t}}$$

Our method

Returning from the recursion:

$T^2 = 4\ 1\ 2\ 4\ 1\ 3\ 2$

BWT^2	
4	1 2
4	1 3 2
1	2
3	2
1	3
2	4 1 2
2	4 1 3 2

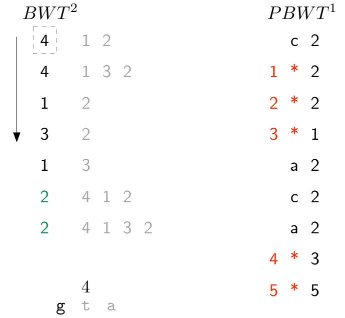
$PBWT^1$		
	c	2
1	*	2
2	*	2
3	*	1
	a	2
	c	2
	a	2
4	*	3
5	*	5

$D^1 = \overset{1}{\texttt{a}\ 5}\quad \overset{2}{*\ \$}\quad \overset{3}{\texttt{a}\ 4}\quad \overset{4}{\texttt{g}\ 5}\quad \overset{5}{*\ \texttt{t}}$

Our method

Returning from the recursion:

$T^2 = 4\ 1\ 2\ 4\ 1\ 3\ 2$



BWT^2			$PBWT^1$	
4	1 2		c	2
4	1 3 2	1	*	2
1	2	2	*	2
3	2	3	*	1
1	3		a	2
2	4 1 2		c	2
2	4 1 3 2		a	2
		4	*	3
		5	*	5

4
g t a

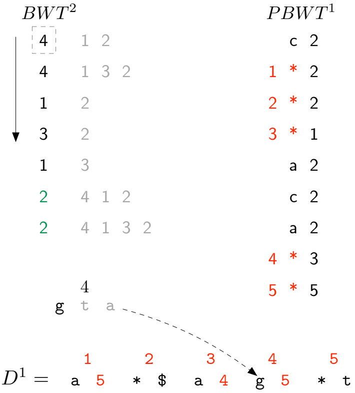
1 2 3 4 5

$D^1 =$ a 5 * $ a 4 g 5 * t

Our method

Returning from the recursion:

$T^2 =$ 4 1 2 4 1 3 2



BWT^2	
4	1 2
4	1 3 2
1	2
3	2
1	3
2	4 1 2
2	4 1 3 2

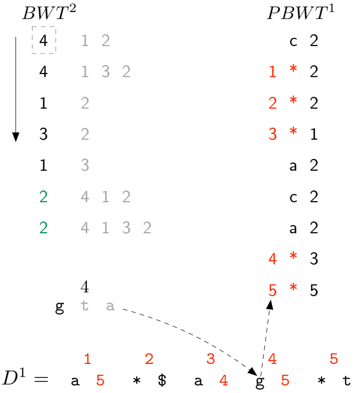
$PBWT^1$		
	c	2
1	*	2
2	*	2
3	*	1
	a	2
	c	2
	a	2
4	*	3
5	*	5

	1	2	3	4	5
$D^1 =$	a 5	* $	a 4	g 5	* t

Our method

Returning from the recursion:

$T^2 =$ 4 1 2 4 1 3 2



BWT^2

$\boxed{4}$ 1 2
4 1 3 2
1 2
3 2
1 3
2 4 1 2
2 4 1 3 2

$PBWT^1$

c 2
1 * 2
2 * 2
3 * 1
a 2
c 2
a 2
4 * 3
5 * 5

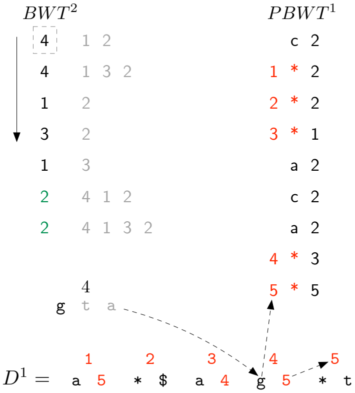
4
g t a

$D^1 =$ a 5 * \$ a 4 g 5 * t

Our method

Returning from the recursion:

$T^2 =$ 4 1 2 4 1 3 2



BWT^2	
4	1 2
4	1 3 2
1	2
3	2
1	3
2	4 1 2
2	4 1 3 2

$PBWT^1$		
	c	2
1	*	2
2	*	2
3	*	1
	a	2
	c	2
	a	2
4	*	3
5	*	5

$D^1 =$ a 5 * $ a 4 g 5 * t

Experiments: datasets

Dataset	σ	n (GB)	n/r
ILL1	5	12.77	3.18
ILL2	5	24.36	4.07
ILL3	5	35.84	4.67
ILL4	5	46.50	5.03
ILL5	5	57.37	5.33
HGA05	16	14.27	4.82
HGA10	16	29.63	8.76
HGA15	16	45.04	12.02
HGA20	16	60.01	15.67
HGA25	16	75.05	19.42

Table: ILLX= Illumina reads. HGAXX= assembled human genomes.

Experiments: competitors

Experiments: competitors

- `ropebwt2`: a variation of the original BCR algorithm of **Bauer et al. 2013** that uses rope data structures.

Experiments: competitors

- `ropebwt2`: a variation of the original BCR algorithm of **Bauer et al. 2013** that uses rope data structures.
- `pfp-eBWT`: the eBWT algorithm of **Boucher et al. 2021** that builds on prefix-free parsing + ISS.

Experiments: competitors

- ropebwt2: a variation of the original BCR algorithm of **Bauer et al. 2013** that uses rope data structures.
- pfp-eBWT: the eBWT algorithm of **Boucher et al. 2021** that builds on prefix-free parsing + ISS.
- BCR_LCP_GSA: the current implementation of the semi-external BCR algorithm.

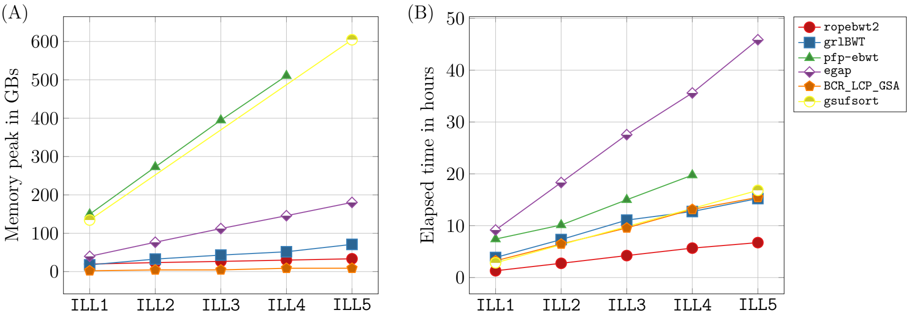
Experiments: competitors

- ropebwt2: a variation of the original BCR algorithm of **Bauer et al. 2013** that uses rope data structures.
- pfp-eBWT: the eBWT algorithm of **Boucher et al. 2021** that builds on prefix-free parsing + ISS.
- BCR_LCP_GSA: the current implementation of the semi-external BCR algorithm.
- egap: a semi-external algorithm of **Edigi et al. 2019** that builds the BCR BWT.

Experiments: competitors

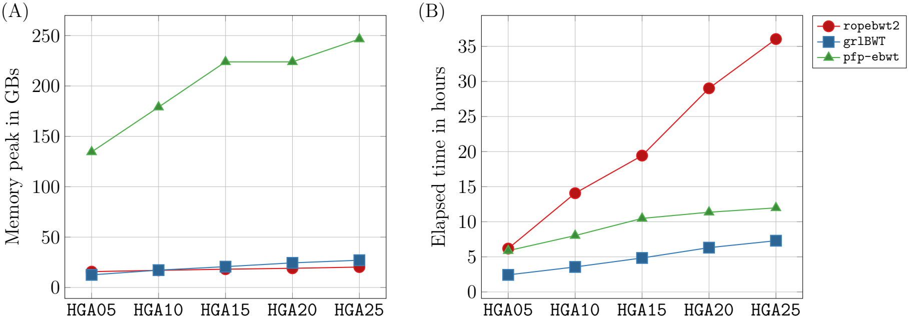
- ropebwt2: a variation of the original BCR algorithm of **Bauer et al. 2013** that uses rope data structures.
- pfp-eBWT: the eBWT algorithm of **Boucher et al. 2021** that builds on prefix-free parsing + ISS.
- BCR_LCP_GSA: the current implementation of the semi-external BCR algorithm.
- egap: a semi-external algorithm of **Edigi et al. 2019** that builds the BCR BWT.
- gsufsort: an in-memory method proposed by **Louza et al. 2020** that computes the BCR BWT and (optionally) other data structures.

Non-repetitive data (Illumina reads)



Experiments: results

Repetitive data (assembled genomes)



Future work

- Extend our procedure to build other data structures: LCP, SA samples (r-index).

Future work

- Extend our procedure to build other data structures: LCP, SA samples (r-index).
- Modify the algorithm to build different BWT variations (e.g., the eBWT).

Future work

- Extend our procedure to build other data structures: LCP, SA samples (r-index).
- Modify the algorithm to build different BWT variations (e.g., the eBWT).
- Improve our hash table implementation.

Future work

- Extend our procedure to build other data structures: LCP, SA samples (r-index).
- Modify the algorithm to build different BWT variations (e.g., the eBWT).
- Improve our hash table implementation.
- Use our repetition-aware strategy to perform other calculations: MEMs, MUMs, or suffix-prefix overlaps.

Questions?