Efficient Construction of the BWT for Repetitive Text Using String Compression

Diego Díaz\textsuperscript{1}  
Gonzalo Navarro\textsuperscript{2}

\textsuperscript{1}Department of Computer Science, University of Helsinki

\textsuperscript{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.
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 ...
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:**
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.
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*. 
Let $\mathcal{T} = \{ T_1, T_2, \ldots, T_k \}$ be a string collection of $k$ strings and $n = \sum_{1}^{k} |T_i|$ symbols.
Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of $k$ strings and $n = \sum_{1}^{k} |T_i|$ symbols.

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 $n = \sum_{i=1}^{k} |T_i|$ symbols.

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:
Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of $k$ strings and $n = \Sigma_{i=1}^{k} |T_i|$ 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).
Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of $k$ strings and $n = \sum_{1}^{k}|T_i|$ symbols.

**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) $t \ c \ \ldots$
  - (B) $t \ t \ t \ c \ \ldots$
  - $L \ \ldots$
  - $L \ L \ L \ L$

- 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 \((S)\):
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 ($S$):

\[
\begin{align*}
\text{(A)} & \quad c \quad t \quad \ldots \\
\text{S} & \\
\text{(B)} & \quad c \quad c \quad c \quad t \quad \ldots \\
\text{S} & \quad S \quad S \quad S
\end{align*}
\]
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 ($S^*$):
  
  ```
  L S^* S
  ```

```c
caaa...
```
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:
  - $t \ c \ g \ g \ t \ a \ g \ \ldots$
  - $L \ S^* \ S \ S \ L \ S^* \ L$
Our method

Input:
Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a string collection of $k$ strings and $n = \sum_{1}^{k} |T_i|$ 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 $n = \sum_{1}^{k} |T_i|$ symbols for which we require to build the BCR BWT.

**Observation**
Let $S$ be the set of distinct strings of length $> 1$ appearing as suffixes in the LMS substrings of $\mathcal{T}$. $S$ 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 $S$ be the set of distinct strings of length $> 1$ appearing as suffixes in the LMS substrings of $\mathcal{T}$. $S$ 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 $Y \in S$ appear consecutively in the suffix array.
Our method

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

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

<table>
<thead>
<tr>
<th>BWT</th>
<th>SA (of $T$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>. a c t g L g a c ...</td>
<td></td>
</tr>
<tr>
<td>X . a c t g L g a c ...</td>
<td></td>
</tr>
<tr>
<td>. a c t g L g a c ...</td>
<td></td>
</tr>
<tr>
<td>. a c t g_s* t ...</td>
<td></td>
</tr>
<tr>
<td>Y . a c t g_s* t ...</td>
<td></td>
</tr>
<tr>
<td>. a c t g_s* t ...</td>
<td></td>
</tr>
</tbody>
</table>

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 $T$ 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 $T$ to fill as many positions in the BWT as possible.

Let $D$ be the set of strings occurring as LMS substrings in $T$. 
Our method

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

Let $D$ be the set of strings occurring as LMS substrings in $T$.

Let $Y = actg \in S$ 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 $T$ to fill as many positions in the BWT as possible.

Let $D$ be the set of strings occurring as LMS substrings in $T$.

Let $Y = actg \in S$ 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 $T$ to fill as many positions in the BWT as possible.

Let $D$ be the set of strings occurring as LMS substrings in $T$.

Let $Y = actg \in S$ 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.
**Our idea:** we use the partition in the SA induced by the LMS substrings of $T$ to fill as many positions in the BWT as possible.

Let $D$ be the set of strings occurring as LMS substrings in $T$.

Let $Y = \text{actg} \in S$ 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 $D$.
Our method

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

Let $D$ be the set of strings occurring as LMS substrings in $T$.

Let $Y = \text{actg} \in S$ 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.
Our method (like ISS) is recursive.
Our method (like ISS) is recursive.

In each recursive step $i$, we proceed as follows:
Our method

Entering the recursion:

\[ T^1 = \text{gtattacc} \quad \$ \quad \text{gtattagg} \quad \$ \]

\[
S \quad L \quad S^* \quad L \quad L \quad S^* \quad L \quad L \quad S^* \quad S \quad L \quad S^* \quad S \quad L \quad S^* \quad L \quad L \quad S^*
\]
Our method

Entering the recursion:

\[ T^1 = \text{gtattacc}\$ \text{gtaataagtaacc}\$ \]
\[ S \ L \ S^* \ L \ L \ S^* \ L \ L \ S^* \ S \ L \ S^* \ S \ L \ S^* \ L \ L \ S^* \]
\[ D^1 = \text{acc}\$ \text{agtaaaataaga} \]
\[ N^1 = \begin{array}{cccccc}
2 & 1 & 2 & 2
\end{array} \]
Our method

Entering the recursion:

\[ T^1 = \text{gtattacc}\$ \quad \text{gtataagtaacc}\$ \]
\[ S \quad L \quad S^* \quad L \quad L \quad S^* \quad L \quad L \quad S^* \quad S \quad L \quad S^* \quad L \quad L \quad S^* \]

\[ D^1 = \text{acc}\$ \quad \text{agtaaataga} \]

\[ N^1 = \begin{bmatrix} 2 & 1 & 2 & 2 \end{bmatrix} \]

\[ SA_{D^1} = \begin{bmatrix} 4 & 9 & 1 & 5 & 10 & 8 & 12 & 15 & 3 & 2 & 6 & 13 & 7 & 11 & 14 \end{bmatrix} \]
Our method

Entering the recursion:

\[ T^1 = \text{gtata}t\text{ata}c\text{cc}\$\text{gtata}tag\text{tata}c\text{cc}\$ \]

\[ D^1 = \text{c}\text{c}\$\text{agt}\text{ata}\text{ata}\text{ata}\text{tg}\text{ta} \]

\[ N^1 = \begin{array}{cccc}
2 & 1 & 2 & 2
\end{array} \]

\[ SA_D^1 = \begin{array}{cccccccccccc}
4 & 9 & 1 & 5 & 10 & 8 & 12 & 15 & 3 & 2 & 6 & 13 & 7 & 11 & 14
\end{array} \]

\[ pBWT^1 = \begin{array}{cccccccc}
\text{c} & * & * & * & a & \text{c} & \text{a} & * & *
\end{array} \]

\[ \begin{array}{cccc}
2 & 2 & 2 & 1 & 2 & 2 & 2 & 3 & 5
\end{array} \]
Our method

Entering the recursion:

\[ T^1 = \text{gtatattacc} \quad \text{gtataatagtatacc} \]

\[ S \quad L \quad S^* \quad L \quad L \quad S^* \quad L \quad L \quad S^* \quad L \quad S^* \quad L \quad L \quad S^* \]

\[ D^1 = \text{ac} \quad \text{c} \quad \text{a} \quad \text{c} \quad \text{a} \quad \text{g} \quad \text{t} \quad \text{a} \quad \text{a} \quad \text{t} \quad \text{a} \quad \text{g} \quad \text{t} \quad \text{a} \]

\[ N^1 = \quad 2 \quad 1 \quad 2 \quad 2 \]

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

\[ pBWT^1 = \text{c} \quad * \quad * \quad * \quad \text{a} \quad \text{c} \quad \text{a} \quad * \quad * \]

\[ aatataattaccc \quad aagttagttat \]

\[ D^1 = \text{a} \quad 5 \quad * \quad \$ \quad \text{a} \quad 4 \quad \text{g} \quad 5 \quad * \quad \text{t} \]
Our method

Entering the recursion:

\[
T^1 = \text{gtatcatch}\text{gtatagtcacc} \\
\]

\[
D^1 = \text{acca}\text{agtaaataaga} \\
N^1 = \begin{bmatrix} 2 & 1 & 2 & 2 \end{bmatrix}
\]

\[
SA_{D^1} = \begin{bmatrix} 4 & 9 & 1 & 5 & 10 & 8 & 12 & 15 & 3 & 2 & 6 & 13 & 7 & 11 & 14 \end{bmatrix}
\]

\[
pBWT^1 = \begin{bmatrix} c & * & * & * & a & c & a & * & * \end{bmatrix} \\
2 2 2 1 2 2 2 3 5
\]

\[
D^1 = \begin{bmatrix} a & 5 & * & $ & a & 4 & g & 5 & * & t \end{bmatrix}
\]

New parse = 4 1 2 4 1 3 2

D. Díaz, G. Navarro
Our method

Returning from the recursion:

\[ T^2 = 4 \ 1 \ 2 \ 4 \ 1 \ 3 \ 2 \]
Our method

Returning from the recursion:

\[ T^2 = 4 \ 1 \ 2 \ 4 \ 1 \ 3 \ 2 \]

\[ BWT^2 \\
\begin{array}{ccc}
4 & 1 & 2 \\
4 & 1 & 3 & 2 \\
1 & 2 \\
3 & 2 \\
1 & 3 \\
2 & 4 & 1 & 2 \\
2 & 4 & 1 & 3 & 2 \\
\end{array} \]
Our method

Returning from the recursion:

\[ T^2 = 4 \ 1 \ 2 \ 4 \ 1 \ 3 \ 2 \]

<table>
<thead>
<tr>
<th>BWT^2</th>
<th>PBWT^1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 1 2</td>
<td>c 2</td>
</tr>
<tr>
<td>4 1 3 2</td>
<td>1 * 2</td>
</tr>
<tr>
<td>1 2</td>
<td>2 * 2</td>
</tr>
<tr>
<td>3 2</td>
<td>3 * 1</td>
</tr>
<tr>
<td>1 3</td>
<td>a 2</td>
</tr>
<tr>
<td>2 4 1 2</td>
<td>c 2</td>
</tr>
<tr>
<td>2 4 1 3 2</td>
<td>a 2</td>
</tr>
<tr>
<td>4 * 3</td>
<td></td>
</tr>
<tr>
<td>5 * 5</td>
<td></td>
</tr>
</tbody>
</table>
Our method

Returning from the recursion:

\[
T^2 = 4 \ 1 \ 2 \ 4 \ 1 \ 3 \ 2
\]

\[
BWT^2 =
\begin{array}{cccc}
4 & 1 & 2 \\
4 & 1 & 3 & 2 \\
1 & 2 \\
3 & 2 \\
1 & 3 \\
2 & 4 & 1 & 2 \\
2 & 4 & 1 & 3 & 2
\end{array}
\]

\[
PBWT^1 =
\begin{array}{c}
c & 2 \\
1 \ * \ 2 \\
2 \ * \ 2 \\
3 \ * \ 1 \\
a \ 2 \\
c \ 2 \\
a \ 2 \\
4 \ * \ 3 \\
5 \ * \ 5
\end{array}
\]

\[
D^1 =
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
a & 5 \ * \ $ \ a & 4 \ g & 5 \ * \ t
\end{array}
\]
Returning from the recursion:

\[
T^2 = 4 \ 1 \ 2 \ 4 \ 1 \ 3 \ 2
\]

\[
\begin{array}{c|c}
BWT^2 & PBWT^1 \\
\hline
4 & 2 \\
4 & 1 \ 3 \ 2 \\
1 & 2 \\
3 & 2 \\
1 & 3 \\
2 & 4 \ 1 \ 2 \\
2 & 4 \ 1 \ 3 \ 2
\end{array}
\]

\[
D^1 = \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
a & 5 & * & $ & a & 4 & g & 5 & * & t
\end{array}
\]
Our method

Returning from the recursion:

\[ T^2 = 4 \ 1 \ 2 \ 4 \ 1 \ 3 \ 2 \]

\[
\begin{array}{c}
BWT^2 \\
\begin{array}{c}
4 \\
4 \\
1 \\
3 \\
1 \\
2 \\
2 \\
4 \\
4 \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
PBWT^1 \\
c \\
1 * 2 \\
2 * 2 \\
3 * 1 \\
a \\
c \\
a \\
4 * 3 \\
5 * 5 \\
g \\
t \\
a
\end{array}
\]

\[
D^1 = \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
a & 5 & * & $ & a & 4 & g & 5 & * & t
\end{array}
\]
Our method

Returning from the recursion:

\[
T^2 = [4 1 2 4 1 3 2]
\]

\[
BWT^2
\]

\[
P_{BWT}^1
\]

\[
D^1 = [1 2 3 4 5]
\]

\[
a 5 * \$ a 4 g 5 * t
\]
Our method

Returning from the recursion:

\[ T^2 = 4 \ 1 \ 2 \ 4 \ 1 \ 3 \ 2 \]

\[
\begin{align*}
BWT^2 & = \begin{array}{cccc}
4 & 1 & 2 \\
4 & 1 & 3 & 2 \\
1 & 2 \\
3 & 2 \\
1 & 3 \\
2 & 4 & 1 & 2 \\
2 & 4 & 1 & 3 & 2 \\
\end{array} \\
PBWT^1 & = \begin{array}{c}
c & 2 \\
1 \ * \ 2 \\
2 \ * \ 2 \\
3 \ * \ 1 \\
a & 2 \\
c & 2 \\
a & 2 \\
4 \ * \ 3 \\
5 \ * \ 5 \\
\end{array}
\]

\[ D^1 = \begin{array}{cccc}
1 & 2 & 3 & 4 & 5 \\
a & 5 & \ * \ & a & 4 & g & 5 & \ * \ & t \\
\end{array} \]
Our method

Returning from the recursion:

\[ T^2 = 4 \ 1 \ 2 \ 4 \ 1 \ 3 \ 2 \]

\[ BWT^2 \]
\[
\begin{array}{c}
4 \\
4 \\
1 \\
3 \\
1 \\
2 \\
3 \\
2 \\
\end{array}
\]

\[ PBWT^1 \]
\[
\begin{array}{c}
c \\
1 \ *
\end{array}
\]

\[ D^1 =
\]
\[
\begin{array}{c}
a \\
5 \\
\$ \\
a \\
g \\
4 \\
5 \ *
\end{array}
\]
## Experiments: datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>( \sigma )</th>
<th>( n ) (GB)</th>
<th>( n/r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILL1</td>
<td>5</td>
<td>12.77</td>
<td>3.18</td>
</tr>
<tr>
<td>ILL2</td>
<td>5</td>
<td>24.36</td>
<td>4.07</td>
</tr>
<tr>
<td>ILL3</td>
<td>5</td>
<td>35.84</td>
<td>4.67</td>
</tr>
<tr>
<td>ILL4</td>
<td>5</td>
<td>46.50</td>
<td>5.03</td>
</tr>
<tr>
<td>ILL5</td>
<td>5</td>
<td>57.37</td>
<td>5.33</td>
</tr>
<tr>
<td>HGA05</td>
<td>16</td>
<td>14.27</td>
<td>4.82</td>
</tr>
<tr>
<td>HGA10</td>
<td>16</td>
<td>29.63</td>
<td>8.76</td>
</tr>
<tr>
<td>HGA15</td>
<td>16</td>
<td>45.04</td>
<td>12.02</td>
</tr>
<tr>
<td>HGA20</td>
<td>16</td>
<td>60.01</td>
<td>15.67</td>
</tr>
<tr>
<td>HGA25</td>
<td>16</td>
<td>75.05</td>
<td>19.42</td>
</tr>
</tbody>
</table>

**Table:** ILL\(X\) = Illumina reads. HGA\(XX\) = assembled human genomes.
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.
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.
- 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.
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.
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.
Experiments: results

Non-repetitive data (Illumina reads)

(A) Memory peak in GBs

(B) Elapsed time in hours

D. Díaz, G. Navarro
Efficient Construction of the BWT
June 27, 2022 12 / 14
Experiments: results

Repetitive data (assembled genomes)

(A)

Memory peak in GBs

(B)

Elapsed time in hours

ropebwt2
griBWT
pfp-ebwt

D. Díaz, G. Navarro
Efficient Construction of the BWT
June 27, 2022 12 / 14
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.
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?