Text Indexing

Lecture 07: FM-Index and r-Index

Florian Kurpicz
Recap: Wavelet Trees
Recap: Wavelet Trees

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$[0, 7]$
Recap: Wavelet Trees

0 1 6 7 1 5 4 2 6 3
0 0 1 1 0 1 1 0 1 0

0 1 1 0 0 1 1 1
0 0 0 0 0 0 1

0 1 6 7 1 5 4 2 6 3
0 0 1 1 0 1 1 0 1 0
0 0 1 1 0 0 0 1 1 1
0 1 0 1 1 0 0 0 1
Recap: Wavelet Trees

\[
\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]
Recap: Wavelet Trees

\[
[0, 7]
\]

\[
\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
[0, 3]
\]

\[
\begin{array}{cccc}
0 & 1 & 1 & 2 & 3 \\
0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

\[
[4, 7]
\]

\[
\begin{array}{cccccc}
6 & 7 & 5 & 4 & 6 \\
1 & 1 & 0 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]
Recap: Wavelet Trees
Recap: Wavelet Trees

\[
\begin{array}{cccccc}
0 & 1 & 6 & 7 & 1 & 5 \\
0 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 1 & 2 \\
0 & 0 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
6 & 7 & 5 & 4 \\
1 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]
Recap: Wavelet Trees
Recap: Wavelet Trees

![Wavelet Tree Diagram]

- [0, 7]
- [0, 3]
- [4, 7]
- [0, 1]
- [2, 3]
- [4, 5]
- [6, 7]

rank\_6(9)
Recap: Wavelet Trees

\[ \begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array} \]

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 0 & 0 & 1 & 1 \\
\end{array} \]

\[ \begin{array}{cccc}
6 & 7 & 5 & 4 & 6 \\
1 & 1 & 0 & 0 & 1 \\
\end{array} \]

\[ \begin{array}{cccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{array} \]

\[ \text{rank}_6(9) = 110 \]
Recap: Wavelet Trees

\[
\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]
Recap: Wavelet Trees

![Wavelet Tree Diagram]

- **[0, 7]**
- **[0, 3]**
- **[4, 7]**
- **[0, 1]**
- **[2, 3]**
- **[4, 5]**
- **[6, 7]**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- **[0, 1]**
- **[2, 3]**
- **[4, 5]**
- **[6, 7]**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>7</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Recap: Wavelet Trees
Recap: Compressed Wavelet Trees

- build wavelet tree for compressed text
- compress text using bit-wise negated canonical Huffman-codes

- intervals are only missing to the right (white space)
- no holes allow for easy querying
Recap: Compressed Wavelet Trees

- build wavelet tree for compressed text
- compress text using bit-wise negated canonical Huffman-codes
- can a wavelet tree be compressed further?

Intervals are only missing to the right (white space)
No holes allow for easy querying
Recap: Compressed Wavelet Trees

- build wavelet tree for compressed text
- compress text using bit-wise negated canonical Huffman-codes
- can a wavelet tree be compressed further?
- PINGO are there compressed bit vectors with $O(1)$ access time?

Intervals are only missing to the right (white space)
No holes allow for easy querying

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
compress (sparse) bit vectors
- bit vector contains \( k \) one bits
- use \( O(k \lg \frac{n}{k}) + o(n) \) bits
- retrieve \( \Theta(\lg n) \) bits at the same time
- similar to \( rank \) data structure
Bit Vector Compression (1/2)

- compress (sparse) bit vectors
- bit vector contains \( k \) one bits
- use \( O(k \lg \frac{n}{k}) + o(n) \) bits
- retrieve \( \Theta(\lg n) \) bits at the same time
- similar to rank data structure

- split bit vector into (super-)blocks
- blocks of size \( s = \frac{\lg n}{2} \)
- super-blocks of size \( s' = s^2 \)
compress (sparse) bit vectors
bit vector contains \( k \) one bits
use \( O(k \lg \frac{n}{k}) + o(n) \) bits
retrieve \( \Theta(\lg n) \) bits at the same time
similar to rank data structure

- split bit vector into (super-)blocks
- blocks of size \( s = \frac{\lg n}{2} \)
- super-blocks of size \( s' = s^2 \)

Array \( C \)
- number of ones in \( i \)-th block
Bit Vector Compression (1/2)

- compress (sparse) bit vectors
- bit vector contains \( k \) one bits
- use \( O(k \log \frac{n}{k}) + o(n) \) bits
- retrieve \( \Theta(\log n) \) bits at the same time
- similar to rank data structure

- split bit vector into (super-)blocks
- blocks of size \( s = \frac{\log n}{2} \)
- super-blocks of size \( s' = s^2 \)

Lookup-Tables \( L_i \)
- for \( i \in [0, s] \) store lookup-table containing all bit vectors with \( i \) one bits

Array \( C \)
- number of ones in \( i \)-th block
Bit Vector Compression (1/2)

- compress (sparse) bit vectors
- bit vector contains \( k \) one bits
- use \( O(k \log \frac{n}{k}) + o(n) \) bits
- retrieve \( \Theta(\log n) \) bits at the same time
- similar to rank data structure

- split bit vector into (super-)blocks
- blocks of size \( s = \frac{\log n}{2} \)
- super-blocks of size \( s' = s^2 \)

Array \( C \)
- number of ones in \( i \)-th block

Lookup-Tables \( L_i \)
- for \( i \in [0, s] \) store lookup-table containing all bit vectors with \( i \) one bits
- use variable-length codes to identify content of block
- concatenate all codes in bit vector \( V \)
Bit Vector Compression (1/2)

- compress (sparse) bit vectors
- bit vector contains \( k \) one bits
- use \( O(k \log \frac{n}{k}) + o(n) \) bits
- retrieve \( \Theta(\log n) \) bits at the same time
- similar to rank data structure

- split bit vector into (super-)blocks
- blocks of size \( s = \frac{\log n}{2} \)
- super-blocks of size \( s' = s^2 \)

Array \( C \)
- number of ones in \( i \)-th block

Lookup-Tables \( L_i \)
- for \( i \in [0, s] \) store lookup-table containing all bit vectors with \( i \) one bits
- use variable-length codes to identify content of block
- concatenate all codes in bit vector \( V \)

Bit Vector \( V \)
- let \( k_i \) be number of ones in \( i \)-th block
- use \( \lceil \log \binom{s}{k_i} \rceil \) bits to encode block position in lookup-table
- concatenate all codes
Bit Vector Compression (2/2)

Array $SBlock$

- for every super-block $i$, $SBlock[i]$ contains position of encoding of first block in $i$-th super-block in $V$
- $\lceil \log n \rceil$ bits per entry
### Array $SBlock$
- for every super-block $i$, $SBlock[i]$ contains position of encoding of first block in $i$-th super-block in $V$
- $\lceil \lg n \rceil$ bits per entry

### Array $Block$
- for every block $i$, $Block[i]$ contains position of encoding of $i$-th block in $V$ relative to its super-block
- $O(\lg \lg n)$ bits per entry
Bit Vector Compression (2/2)

**Array SBlock**
- for every super-block $i$, $SBlock[i]$ contains position of encoding of first block in $i$-th super-block in $V$
- $\lceil \log n \rceil$ bits per entry

**Array Block**
- for every block $i$, $Block[i]$ contains position of encoding of $i$-th block in $V$ relative to its super-block
- $O(\log \log n)$ bits per entry

**Lemma: Compressed Bit Vectors**
A bit vector of size $n$ containing $k$ ones can be represented using $O(k \log \frac{n}{k}) + o(n)$ bits allowing $O(1)$ time access to individual bits
Bit Vector Compression 📝 (2/2)

Array \textit{SBlock}
- for every super-block \( i \), \textit{SBlock}[i] \text{ contains position of encoding of first block in } \( i \)-th
  super-block in \( V \)
- \( \lceil \log n \rceil \) bits per entry

Array \textit{Block}
- for every block \( i \), \textit{Block}[i] \text{ contains position of encoding of } \( i \)-th
  block in \( V \) relative to its
  super-block
- \( O(\log \log n) \) bits per entry

Lemma: Compressed Bit Vectors
A bit vector of size \( n \) containing \( k \) ones can be
represented using \( O(k \log \frac{n}{k}) + o(n) \) bits allowing
\( O(1) \) time access to individual bits

Proof (Sketch space requirements)
- \(|C| = O\left(\frac{n}{s} \log s\right) = o(n) \) bits
- \(|SBlock| = O\left(\frac{n}{s} \log n\right) = o(n) \) bits
- \(|Block| = O\left(\frac{n}{s} \log s\right) = o(n) \) bits
- \( \sum_{k=0}^{s} |L_k| \leq (s + 1)2^s s = o(n) \) bits
- \( |V| = \sum_{i=1}^{\left\lceil \frac{n}{s} \right\rceil} \left\lceil \log \left(\frac{s^i}{k_i}\right) \right\rceil \leq \log \left(\frac{n}{k}\right) + \frac{n}{s} \leq \log((n/k)^k) + \frac{n}{s} = k \log \frac{n}{k} + O\left(\frac{n}{\log n}\right) \) bits
Recap: Backwards Search in the BWT

**Function** `BackwardsSearch(P[1..n], C, rank)`:  
1. \( s = 1, \ e = n \)
2. \( \text{for } i = m, \ldots, 1 \text{ do} \)
3. \( s = C[P[i]] + rank_{P[i]}(s - 1) + 1 \)
4. \( e = C[P[i]] + rank_{P[i]}(e) \)
5. \( \text{if } s > e \text{ then} \)
6. \( \quad \text{return } \emptyset \)
7. \( \text{return } [s, e] \)

- no access to text or `SA` required
- no binary search
- existential queries are easy
- counting queries are easy
- reporting queries require additional information
- example on the board 📌
The FM-Index [FM00]

Building Blocks of FM-Index
- wavelet tree on BWT providing \textit{rank}-function
- \(C\)-array
- sampled suffix array with sample rate \(s\)
- bit vector marking sampled suffix array positions

Lemma: FM-Index
Given a text \(T\) of length \(n\) over an alphabet of size \(\sigma\), the FM-index requires \(O(n \lg \sigma)\) bits of space and can answer counting queries in \(O(m \lg \sigma)\) time and reporting queries in \(O(\text{occ} + m \lg \sigma)\) time
The FM-Index [FM00]

Building Blocks of FM-Index
- wavelet tree on BWT providing rank-function
- C-array
- sampled suffix array with sample rate $s$
- bit vector marking sampled suffix array positions

Space Requirements
- wavelet tree: $n\lceil\lg \sigma\rceil(1 + o(1))$ bits
- C-array: $\sigma\lceil\lg n\rceil$ bits $\Theta n(1 + o(1))$ bits if $\sigma \geq \frac{n}{\lg n}$
- sampled suffix array: $\frac{n}{s}\lceil\lg n\rceil$ bits
- bit vector: $n(1 + o(1))$ bits

Lemma: FM-Index
Given a text $T$ of length $n$ over an alphabet of size $\sigma$, the FM-index requires $O(n\lg \sigma)$ bits of space and can answer counting queries in $O(m\lg \sigma)$ time and reporting queries in $O(o_c + m\lg \sigma)$ time.

space and time bounds can be achieved with $s = \lg_\sigma n$
Conclusion FM-Index

- FM-index is easy to compress
- wavelet tree on $BWT$ can be compressed
- bit vector can be compressed

- very small in comparison with suffix tree or suffix array
- compression does not make use of structure of $BWT$ waveset trees are compressed using Huffman-codes

Definition: Run (simplified, recap)

Given a text $T$ of length $n$, we call its substring $T[i..j]$ a run, if $T[k] = T[\ell]$ for all $k, \ell \in [i, j]$ and $T[i-1] \neq T[i]$ and $T[j+1] \neq T[j]$.

(To be more precise, this is a definition for a run of a periodic substring with smallest period 1, but this is not important for this lecture.)
Conclusion FM-Index

- FM-index is easy to compress
- wavelet tree on $BWT$ can be compressed
- bit vector can be compressed
- very small in comparison with suffix tree or suffix array
- compression does not make use of structure of $BWT$
- wavelet trees are compressed using Huffman-codes

Definition: Run (simplified, recap)

Given a text $T$ of length $n$, we call its substring $T[i..j]$ a run, if

- $T[k] = T[\ell]$ for all $k, \ell \in [i, j]$ and
- $T[i - 1] \neq T[i]$ and $T[j + 1] \neq T[j]$

(To be more precise, this is a definition for a run of a periodic substring with smallest period 1, but this is not important for this lecture.)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>a</td>
<td>b</td>
<td>$</td>
<td>$</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>
Motivation: r-Index

- next: compressed index
- how to measure compressibility?
Motivation: \( r \)-Index

- next: compressed index
- how to measure compressibility?

Measure for Compressibility

- \( k \)-th order empirical entropy \( H_k \)
- number of LZ factors \( z \)
- number of \( BWT \) runs \( r \)
Motivation: \( r \)-Index

- next: compressed index
- how to measure compressibility?

Measure for Compressibility

- \( k \)-th order empirical entropy \( H_k \)
- number of LZ factors \( z \)
- number of \( BWT \) runs \( r \)

- \( z \) and \( r \) not blind to repetitions
- how do they relate?

Lemma:

Given a text \( T \) of length \( n \). Let \( z \) be the number of LZ77 factors and \( r \) the number of runs in \( T \)'s BWT, then

\[ r \in O(z \lg 2 n) \]

more details in next lecture
Motivation: \( r\)-Index

- next: compressed index
- how to measure compressibility?

Measure for Compressibility

- \( k\)-th order empirical entropy \( H_k \)
- number of LZ factors \( z \)
- number of BWT runs \( r \)

- \( z \) and \( r \) not blind to repetitions
- how do they relate?

Lemma: BWT runs and LZ factors [KK20]

Given a text \( T \) of length \( n \). Let \( z \) be the number of LZ77 factors and \( r \) the number of runs in \( T \)'s BWT, then

\[ r \in O(z \log^2 n) \]
Motivation: $r$-Index

- next: compressed index
- how to measure compressibility?

Measure for Compressibility

- $k$-th order empirical entropy $H_k$
- number of LZ factors $z$
- number of BWT runs $r$

- $z$ and $r$ not blind to repetitions
- how do they relate?

Lemma: BWT runs and LZ factors [KK20]

Given a text $T$ of length $n$. Let $z$ be the number of LZ77 factors and $r$ the number of runs in $T$'s BWT, then

$$r \in O(z \log^2 n)$$

- more details in next lecture
Function BackwardsSearch( P[1..n], C, rank):
1  \( s = 1, \ e = n \)
2  \( \text{for} \ i = m, \ldots, 1 \ \text{do} \)
3       \( s = C[P[i]] + rank_{P[i]}(s - 1) + 1 \)
4       \( e = C[P[i]] + rank_{P[i]}(e) \)
5     \text{if} \ s > e \text{ then} \)
6       \text{return} \emptyset \)
7  \text{return} [s, e]

Goals

- simulate BWT and rank on BWT in
- \( O(r \lg n) \) bits of space
The $r$-Index [GNP20] (1/3)

Given a text $T$ of length $n$ over an alphabet $\Sigma$ and its $BWT$, the $r$-index of this text consists of the following data structures.

- Array $I[i]$ stores the position of the $i$-th run in the $BWT$.
- Array $L'[i]$ stores the character of the $i$-th run in the $BWT$.
- Build a wavelet tree for $L'$.
- Array $R$ stores the lengths of the $BWT$ runs stably sorted by their characters.
- Perform exclusive prefix sum over run lengths.
- Array $C'[\alpha]$ stores the start of the run lengths in $R$ for each character $\alpha \in \Sigma$ starting at 0.
- Bit Vector $B$: compressed bit vector of length $n$ containing ones at positions where $BWT$ runs start.

Rank-supported $r$-Index

2022-12-12 Florian Kurpicz | Text Indexing | 07 FM-Index & $r$-Index

Institute for Theoretical Informatics, Algorithm Engineering
The $r$-Index [GNP20] (1/3)

Given a text $T$ of length $n$ over an alphabet $\Sigma$ and its $BWT$, the $r$-index of this text consists of the following data structures:

- **Array $I$**
  - $I[i]$ stores position of $i$-th run in $BWT$
The $r$-Index [GNP20] (1/3)

Given a text $T$ of length $n$ over an alphabet $\Sigma$ and its $BWT$, the $r$-index of this text consists of the following data structures:

**Array $I$**
- $I[i]$ stores position of $i$-th run in $BWT$

**Array $L'$**
- $L'[i]$ stores character of $i$-th run in $BWT$
- build wavelet tree for $L'$
The $r$-Index \cite{GNP20} (1/3)

Given a text $T$ of length $n$ over an alphabet $\Sigma$ and its BWT, the $r$-index of this text consists of the following data structures:

Array $I$
- $I[i]$ stores position of $i$-th run in BWT

Array $L'$
- $L'[i]$ stores character of $i$-th run in BWT
- build wavelet tree for $L'$

Array $R$
- lengths of BWT runs stably sorted by runs’ characters
- accumulate for each character by performing exclusive prefix sum over run lengths’
The $r$-Index \[\text{[GNP20]}\] (1/3)

Given a text $T$ of length $n$ over an alphabet $\Sigma$ and its $BWT$, the $r$-index of this text consists of the following data structures:

**Array $I$**
- $I[i]$ stores position of $i$-th run in $BWT$

**Array $L'$**
- $L'[i]$ stores character of $i$-th run in $BWT$
- build wavelet tree for $L'$

**Array $R$**
- lengths of $BWT$ runs stably sorted by runs’ characters
- accumulate for each character by performing exclusive prefix sum over run lengths’

**Array $C'$**
- $C'[\alpha]$ stores the start of the run lengths in $R$ for each character $\alpha \in \Sigma$ starting at 0
Given a text $T$ of length $n$ over an alphabet $\Sigma$ and its $BWT$, the $r$-index of this text consists of the following data structures:

**Array $I$**
- $I[i]$ stores position of $i$-th run in $BWT$

**Array $L'$**
- $L'[i]$ stores character of $i$-th run in $BWT$
- build wavelet tree for $L'$

**Array $R$**
- lengths of $BWT$ runs stably sorted by runs’ characters
- accumulate for each character by performing exclusive prefix sum over run lengths’

**Array $C'$**
- $C'[\alpha]$ stores the start of the run lengths in $R$ for each character $\alpha \in \Sigma$ starting at 0

**Bit Vector $B$**
- compressed bit vector of length $n$ containing ones at positions where $BWT$ runs start and rank-support
The \( \text{r-Index} \) (2/3)

\begin{itemize}
\item \( \text{rank}_\alpha (BWT, i) \) with \( \text{r-Index} \)
\item compute number \( j \) of run \( (j = \text{rank}_1 (B, i)) \)
\item compute position \( k \) in \( R \) \( (k = C'[\alpha]) \)
\item compute number \( \ell \) of \( \alpha \) runs before the \( j \)-th run
  \( (\ell = \text{rank}_\alpha (L', j - 1)) \)
\item compute number \( k \) of \( \alpha \)s before the \( j \)-th run
  \( (k = R[k + \ell]) \)
\item compute character \( \beta \) of run \( (\beta = L'[j]) \)
\item if \( \alpha \neq \beta \) return \( k \) \( i \) is not in the run
\item else return \( k + i - l[j] + 1 \) \( i \) is in the run
\end{itemize}
The \( r \)-Index (3/3)

**Lemma: Space Requirements \( r \)-Index**

Given a text \( T \) of length \( n \) over an alphabet of size \( \sigma \) that has \( r \) BWT runs, then its \( r \)-index requires

\[
O(r \lg n) \text{ bits}
\]

and can answer rank-queries on the BWT in \( O(\lg \sigma) \).

Given a pattern of length \( m \), the \( r \)-index can answer pattern matching queries in time

\[
O(m \lg \sigma)
\]
Lemma: Space Requirements $r$-Index

Given a text $T$ of length $n$ over an alphabet of size $\sigma$ that has $r$ BWT runs, then its $r$-index requires $O(r \lg n)$ bits and can answer rank-queries on the BWT in $O(\lg \sigma)$. Given a pattern of length $m$, the $r$-index can answer pattern matching queries in time $O(m \lg \sigma)$.

what about reporting queries?
Locating Occurrences (Sketch)

- modify backwards-search that it maintains \( SA[e] \)
- after backwards-search output \( SA[e], SA[e - 1], \ldots, SA[s] \)
- in \( O(r \lg n) \) bits and \( O(occ \cdot \lg \lg r) \) time

Maintaining \( SA[e] \)

- sample \( SA \) positions at ends of runs
- if next character is \( BWT[e] \), then next \( SA[e'] \) is \( SA[e] - 1 \)
- otherwise locate end of run and extract sample

Output Result

- following \( LF \) not possible \( \approx \) unbounded
- deduce \( SA[i - 1] \) from \( SA[i] \)
- character in \( L \) and \( F \) in same order
- only beginning of runs complicated
- for every character build predecessor data structure over sampled \( SA \)-values at end of runs
- associate with \( \langle i, SA[i] \rangle \)
 Locating Occurrences (Sketch)

- modify backwards-search that it maintains \(SA[e]\)
- after backwards-search output \(SA[e], SA[e - 1], \ldots, SA[s]\)
- in \(O(r \lg n)\) bits and \(O(\text{occ} \cdot \lg \lg r)\) time

Maintaining \(SA[e]\)

- sample \(SA\) positions at ends of runs
- if next character is \(BWT[e]\), then next \(SA[e']\) is \(SA[e] - 1\)
- otherwise locate end of run and extract sample

Output Result

- following \(LF\) not possible \(\S\) unbounded
- deduce \(SA[i - 1]\) from \(SA[i]\)
- character in \(L\) and \(F\) in same order
- only beginning of runs complicated
- for every character build predecessor data structure over sampled \(SA\)-values at end of runs
- associate with \(\langle i, SA[i] \rangle\)

PINGO why can’t we sample the \(SA\) as we did in the FM-index?
From the Suffix Tree to the $r$-Index—Questions?
From the Suffix Tree to the \( r \)-Index—Questions?

- **Suffix Tree**: 1973
- **Suffix Array**: 1993
- **LCP Array**: 1993
From the Suffix Tree to the $r$-Index—Questions?
From the Suffix Tree to the $r$-Index—Questions?

- **Suffix Tree**
  - 1973
  - Memory Requirements

- **Suffix Array**
  - 1993

- **LCP Array**
  - 1993

- **BWT**
  - 1994

- **Wavelet Tree**
  - 2000

- **FM-Index**
  - 2000

- **r-Index**
  - 2018

- **String-Sorting**
  - LCE-Anfragen (Patricia-)Tries
  - Bit-Vektoren mit Rank/Select-Anfragen

- **EM Hashing**
  - Succincte Datenstrukturen

- **Compression**

---

From the Suffix Tree to the $r$-Index—Questions?

1. What are the key differences between the Suffix Tree and the Suffix Array?
2. How do the LCP Array and BWT complement each other in text indexing?
3. What are the advantages of using a Wavelet Tree over a Suffix Tree?
4. How does the FM-Index improve upon traditional suffix structures?
5. What is the role of the $r$-Index in modern text indexing algorithms?
Bibliography I

