Recap: Wavelet Trees

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

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

\[\text{rank}_6(9)\]

\[110\]
Recap: Compressed Wavelet Trees

<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>

- 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
- can a wavelet tree be compressed further?
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
Array \textit{SBlock}
- for every super-block \( i \), \( SBlock[i] \) contains position of encoding of first block in \( i \)-th super-block in \( V \)
- \([\log n]\) bits per entry

Array \textit{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

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
- \(|V| = \sum_{i=1}^{\left\lfloor \frac{n}{s} \right\rfloor} \left\lfloor \log \binom{s}{k_i} \right\rfloor \leq \log \binom{n}{k} + \frac{n}{s} \leq \log((n/k)^k) + \frac{n}{s} = k \log \frac{n}{k} + O\left(\frac{n}{\log n}\right) \) bits
Recap: Backwards Search in the BWT

```
Function BackwardsSearch(P[1..n], C, rank):
  1  s = 1, e = n
  2  for i = m, ..., 1 do
  3     s = C[P[i]] + rank_{P[i]}(s - 1) + 1
  4     e = C[P[i]] + rank_{P[i]}(e)
  5     if s > e then
  6        return ∅
  7  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 📧
### Building Blocks of FM-Index

- wavelet tree on BWT providing \( \text{rank} \)-function
- \( C \)-array
- sampled suffix array with sample rate \( s \)
- bit vector marking sampled suffix array positions

### Space Requirements

- wavelet tree: \( n \lfloor \lg \sigma \rfloor (1 + o(1)) \) bits
- \( C \)-array: \( \sigma \lfloor \lg n \rfloor \) bits \( \oplus \) \( n(1 + o(1)) \) bits if \( \sigma \geq \frac{n}{\lg n} \)
- sampled suffix array: \( \frac{n}{s} \lfloor \lg n \rfloor \) 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 \( O(\text{occ} + m \lg \sigma) \) time

\[ s = \lg_{\sigma} n \]
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 $BWT$ 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.)
Measures of Repetitiveness (Excerpt)

- \( b \) \# phrasess in bidirectional macro scheme
- \( z \) \# phrasess in Lempel-Ziv parsed context-free grammar
- \( g \) size of smallest Suffix-Wheel transform
- \( r \) \# of runs in Burrows-Wheeler transform
- \( e \) \# of nodes and edges in CDAWG

- NP-hard
- \( O(n) \) time
- NP-hard*
- \( O(n) \) time
- \( O(n) \) time

* there are good heuristics
Motivation: $r$-Index

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
Main Part of Backwards-Search

Function \( \text{BackwardsSearch}(P[1..n], C, \text{rank}) \):

1. \( s = 1, e = n \)

2. for \( i = m, \ldots, 1 \) do

3. \[ s = C[P[i]] + \text{rank}_{P[i]}(s - 1) + 1 \]

4. \[ e = C[P[i]] + \text{rank}_{P[i]}(e) \]

5. if \( s > e \) then

6. \quad return \( \emptyset \)

7. 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 $I$**
- $I[i]$ 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 $r$-Index (2/3)

**rank$_{(BWT, i)}$ with $r$-Index**

- Compute number $j$ of run ($j = \text{rank}_1(B, i)$)
- Compute position $k$ in $R$ ($k = C'[\alpha]$)
- Compute number $\ell$ of $\alpha$ runs before the $j$-th run ($\ell = \text{rank}_\alpha(L', j - 1)$)
- Compute number $k$ of $\alpha$s before the $j$-th run ($k = R[k + \ell]$)
- Compute character $\beta$ of run ($\beta = L'[j]$)
- If $\alpha \neq \beta$ return $k \odot i$ is not in the run
- Else return $k + i - I[j] + 1 \odot i$ is in the run
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

$$O(m \lg \sigma)$$

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 if 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 $\langle i, SA[i] \rangle$
The Move Data Structure [NT21]

Definition: Disjoint Interval Sequence

Let \( I = (p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k) \) be a sequence of \( k \) pairs of integers. We introduce a permutation \( \pi \) of \([1, k]\) and sequence \( d_1, d_2, \ldots, d_k \) for \( I \). \( \pi \) satisfies \( q_{\pi[1]} \leq q_{\pi[2]} \leq \cdots \leq q_{\pi[k]} \), and \( d_i = p_{i+1} - p_i \) for \( i \in [1, k] \), where \( p_{k+1} = n + 1 \). We call the sequence \( I \) a disjoint interval sequence if it satisfies the following three conditions:

- \( p_1 = 1 < p_2 < \cdots < p_k \leq n \)
- \( q_{\pi[1]} = 1 \),
- \( q_{\pi[i]} = q_{\pi[i-1]} + d_{\pi[i-1]} \) for each \( i \in [2, k] \).

Move Query

\[
\text{move}(i, x) = (i', x')
\]

- \( i \) position in input interval
- \( x \) input interval
- \( i' \) position in output interval
- \( x' \) input interval covering \( i' \)
From the Suffix Tree to the $r$-Index—Questions?

- **Suffix Tree**: 1973
- **Suffix Array**: 1993
- **BWT**: 1994
- **LCP Array**: 1993
- **Wavelet Tree**: 2000
- **FM-Index**: 2000
- **r-Index**: 2018

Memory Requirements:
- Suffix Tree
- Suffix Array
- BWT
- LCP Array
- Wavelet Tree
- FM-Index
- r-Index

Bit-Vektoren mit Rank/Select-Anfragen

EM Hashing

String-Sorting

LCE-Anfragen

(Patricia-)Tries

Succincte Datenstrukturen

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


