Text Indexing

Lecture 06: Wavelet Trees

Florian Kurpicz
Recap: Rank-Queries

- for a bit vector of size $n$
- blocks of size $s = \left\lfloor \frac{\lg n}{2} \right\rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$
- information for 0s or 1s enough
  \[rank_1(i) = i - rank_0(i)\]
Recap: Rank-Queries

- for a bit vector of size $n$
- blocks of size $s = \left\lfloor \frac{\lg n}{2} \right\rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$
- information for 0s or 1s enough
  \[ rank_1(i) = i - rank_0(i) \]
- for all $\left\lfloor \frac{n}{s'} \right\rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ bits of space
Recap: Rank-Queries

- for a bit vector of size \( n \)
- blocks of size \( s = \left\lfloor \frac{\lg n}{2} \right\rfloor \)
- super blocks of size \( s' = s^2 = \Theta(\lg^2 n) \)
- information for 0s or 1s enough
  \( \rank_1(i) = i - \rank_0(i) \)

- for all \( \left\lfloor \frac{n}{s} \right\rfloor \) blocks, store number of 0s from beginning of super block to end of block
- \( n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n) \) bits of space

- for all \( \left\lfloor \frac{n}{s'} \right\rfloor \) super blocks, store number of 0s from beginning of bit vector to end of super-block
- \( n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n) \) bits of space
Recap: Rank-Queries

- for a bit vector of size $n$
  - blocks of size $s = \left\lfloor \frac{\lg n}{2} \right\rfloor$
  - super blocks of size $s' = s^2 = \Theta(\lg^2 n)$
  - information for 0s or 1s enough
    \[ rank_1(i) = i - rank_0(i) \]

- for all $\left\lfloor \frac{n}{s'} \right\rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
  - $n/s' \cdot \lg n = O(\frac{n \cdot \lg \lg n}{\lg n}) = o(n)$ bits of space

- for all $\left\lfloor \frac{n}{s} \right\rfloor$ blocks, store number of 0s from beginning of super block to end of block
  - $n/s \cdot \lg s' = O(\frac{n \cdot \lg \lg n}{\lg n}) = o(n)$ bits of space

- for all length-$s$ bit vectors, for every position $i$
  - store number of 0s up to $i$
  - $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \cdot \lg n \cdot \lg \lg n) = o(n)$ bits of space
Recap: Rank-Queries

- for a bit vector of size $n$
- blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$
- information for 0s or 1s enough
  - $\text{rank}_1(i) = i - \text{rank}_0(i)$

- for all $\lfloor \frac{n}{s'} \rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
  - $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ bits of space

- for all length-$s$ bit vectors, for every position $i$
  - store number of 0s up to $i$
  - $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n)$ bits of space

- query in $O(1)$ time using three subqueries
  - one in super-block
  - one in block
  - one for remaining bitvector smaller than $s$

- for all $\lfloor \frac{n}{s} \rfloor$ blocks, store number of 0s from beginning of super block to end of block
  - $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ bits of space
Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size $n$ that contains $k$ zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store $k$ solutions in $S[1..k]$ and $select_0(i) = S[i]$ if $k \in O(n/\log n)$ this suffice
Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size $n$ that contains $k$ zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store $k$ solutions in $S[1..k]$ and $select_0(i) = S[i]$ if $k \in O(n/\lg n)$ this suffice

- better: $k/b$ variable-sized super-blocks $B_i$, such that super-block contains $b = \lg^2 n$ zeros
- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}; j - (\lfloor i/b \rfloor b))$
Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size $n$ that contains $k$ zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store $k$ solutions in $S[1..k]$ and
    $select_0(i) = S[i]$ if $k \in O(n/\lg n)$ this suffice

- better: $k/b$ variable-sized super-blocks $B_i$, such that super-block contains $b = \lg^2 n$ zeros
- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}; j - (\lfloor i/b \rfloor b))$

- storing all possible results for the (prefix) sum
- $O((k \lg n)/b) = o(n)$ bits of space
Select in $o(n)$ Space and $O(1)$ Time

- **select** in a bit vector of size $n$ that contains $k$ zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store $k$ solutions in $S[1..k]$ and $select_0(i) = S[i]$ if $k \in O(n/\lg n)$ this suffice

- better: $k/b$ variable-sized super-blocks $B_i$, such that super-block contains $b = \lg^2 n$ zeros
- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b))$

- storing all possible results for the (prefix) sum
- $O((k \lg n)/b) = o(n)$ bits of space

- select on block depends on size of block
  - storing all possible results for the (prefix) sum
  - $O((k \lg n)/b) = o(n)$ bits of space

- storing all possible results for the (prefix) sum
  - $O((k \lg n)/b) = o(n)$ bits of space

- select on block depends on size of block
  - storing all possible results for the (prefix) sum
  - $O((k \lg n)/b) = o(n)$ bits of space

- storing all possible results for the (prefix) sum
  - $O((k \lg n)/b) = o(n)$ bits of space

- select on block depends on size of block
  - storing all possible results for the (prefix) sum
  - $O((k \lg n)/b) = o(n)$ bits of space
Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size $n$ that contains $k$ zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store $k$ solutions in $S[1..k]$ and $select_0(i) = S[i]$ if $k \in O(n/\lg n)$ this suffice

better: $k/b$ variable-sized super-blocks $B_i$, such that super-block contains $b = \lg^2 n$ zeros

- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}; j - (\lfloor i/b \rfloor b))$

- storing all possible results for the (prefix) sum
  - $O((k \lg n)/b) = o(n)$ bits of space

- select on block depends on size of block
  - $|B_{\lfloor i/b \rfloor}| \geq \lg^4 n$: store answers naively
    - requires $O(b \lg n) = O(\lg^3 n)$ bits of space
    - there are at most $O(n/\lg^4 n)$ such blocks
    - total $O(n/\lg n) = o(n)$ bits of space
Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size $n$ that contains $k$ zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store $k$ solutions in $S[1..k]$ and
    
    $select_0(i) = S[i] \uparrow$ if $k \in O(n/\lg n)$ this suffice

- better: $k/b$ variable-sized super-blocks $B_i$, such that super-block contains $b = \lg^2 n$ zeros
  
  $select_0(i) = \sum_{j=0}^{[i/b]-1} |B_j| + select_0(B_{[i/b]}, j - ([i/b]b))$

- storing all possible results for the (prefix) sum
  
  $O((k \lg n)/b) = o(n)$ bits of space

- select on block depends on size of block
  
  $|B_{[i/b]}| \geq \lg^4 n$: store answers naively
  - requires $O(b \lg n) = O(\lg^3 n)$ bits of space
  - there are at most $O(n/\lg^4 n)$ such blocks
  - total $O(n/\lg n) = o(n)$ bits of space

  $|B_{[i/b]}| < \lg^4 n$: divide super-block into blocks
  - same idea: variable-sized blocks containing $b’ = \sqrt{\lg n}$ zeros
  - (prefix) sum $O((k \lg n)/b’) = o(n)$ bits
  - if size $\geq \lg n$ store all answers
  - if size $< \lg n$ store lookup table
Lemma: Binary Rank- and Select-Queries

Given a bit vector of size $n$, there exists data structures that can be computed in time $O(n)$ of size $o(n)$ bits that can answer rank and select queries on the bit vector in $O(1)$ time.
Preliminaries

Definition: Bit Representation

Given a text $T$ over an alphabet of size $\sigma$, each character can be represented using $\lceil \log_2 \sigma \rceil$ bits.

- the leftmost bit is the **most significant bit** and
- the rightmost bit is the **least significant bit**

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

**MSB**

**LSB**
Preliminaries

Definition: Bit Representation

Given a text $T$ over an alphabet of size $\sigma$, each character can be represented using $\lceil \lg \sigma \rceil$ bits.

- the leftmost bit is the **most significant bit** and
- the rightmost bit is the **least significant bit**

- for simplicity characters are integers
- bit representation is integer in binary

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

**Definition: Bit Representation**
Given a text $T$ over an alphabet of size $\sigma$, each character can be represented using $\lceil \lg \sigma \rceil$ bits.
- the leftmost bit is the **most significant bit** and
- the rightmost bit is the **least significant bit**

- for simplicity characters are integers
- bit representation is integer in binary

**Definition: Bit Prefix**
A bit prefix of length $k$ are the $k$ MSBs of a character's bit representation

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MSB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LSB</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Definition: Wavelet Tree

Given a text $T$ of length $n$ over an alphabet $\Sigma = [1, \sigma]$, a wavelet tree is a binary tree, where:

- each node represents characters in $[\ell, r] \subseteq [1, \sigma]$,
- if a node represents characters in $[\ell, r]$, then its left and right child represent characters in $[\ell, (\ell + r)/2)$ and $[(\ell + r)/2, r]$,
- a node is a leaf if $\ell + 2 \geq r$,
- characters are represented using a bit vector,
- an entry is 1 if the character is represented in the right child and 0 otherwise.
Wavelet Trees [GGV03] (1/2)

**Definition: Wavelet Tree**
Given a text $T$ of length $n$ over an alphabet $\Sigma = [1, \sigma]$, a wavelet tree is a binary tree, where
- each node represents characters in $[\ell, r] \subseteq [1, \sigma]$,
- if a node represents characters in $[\ell, r]$, then its left and right child
- represent characters in $[\ell, (\ell + r)/2)$ and $[(\ell + r)/2, r]$
- a node is a leaf if $\ell + 2 \geq r$
- characters are represented using a bit vector
- an entry is 1 if the character is represented in the right child and 0 otherwise

**Definition: Level-wise Wavelet Tree**
A wavelet tree, where all bit vectors on the same depth in the tree are concatenated is called level-wise wavelet tree
Wavelet Trees \cite{GGV03} (1/2)

**Definition: Wavelet Tree**

Given a text $T$ of length $n$ over an alphabet $\Sigma = [1, \sigma]$, a **wavelet tree** is a binary tree, where

- each node represents characters in $[\ell, r] \subseteq [1, \sigma]$,
- if a node represents characters in $[\ell, r]$, then its left and right child
- represent characters in $[\ell, (\ell + r)/2)$ and $[((\ell + r)/2, r]$
- a node is a leaf if $\ell + 2 \geq r$
- characters are represented using a bit vector
- an entry is 1 if the character is represented in the right child and 0 otherwise

**Definition: Level-wise Wavelet Tree**

A wavelet tree, where all bit vectors on the same depth in the tree are concatenated is called **level-wise wavelet tree**

- in practice, level-wise wavelet trees have less overhead
- navigation still easy
Wavelet Trees (2/2)

\[ [0, 7] \]

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

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

\[ [0, 7] \]

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

[0, 7]

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

[0, 3] → [4, 7]

0 1 1 2 3
0 0 0 1 1

6 7 5 4 6
1 1 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 1 0 0 0 1
Wavelet Trees (2/2)

Wavelet Trees (2/2)

Wavelet Trees (2/2)

Wavelet Trees (2/2)

Wavelet Trees (2/2)
Wavelet Trees (2/2)

[0, 7]

[0, 3]

[4, 7]

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

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

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

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

Wavelet Trees (2/2)
Wavelet Trees (2/2)

Wavelet Trees (2/2)
Wavelet Trees (2/2)

Wavelet Trees are a data structure used for efficient rank queries on bit vectors. Each node in a wavelet tree represents a segment of the input bit vector. The tree is constructed such that each level corresponds to a power of 2, and the leaves represent individual bits of the input vector.

The figure illustrates a wavelet tree for the bit vector [0, 7]. The tree is divided into segments [0, 3] and [4, 7], with further divisions into [0, 1], [2, 3], [4, 5], and [6, 7]. The rank function, rank₆(9), is demonstrated by counting the number of 1s up to the 9th position in the bit vector.

```
0 1 6 7 1 5 4 2 6 3
0 0 1 1 0 1 1 0 1 0
0 0 1 0 1 1 0 0 1 1
0 1 0 1 1 1 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 1 0 0 0 1
```
Wavelet Trees (2/2)

Wavelet Trees are a data structure that can efficiently answer range sum queries on a given array. They are particularly useful for handling large datasets that can be represented in binary form.

The diagram above illustrates a Wavelet Tree for an array of numbers. Each level of the tree represents a bit of the binary representation of the numbers in the array. The leaf nodes of the tree correspond to the elements of the array, and the internal nodes represent ranges of elements.

The example shown is for the range sum query \( \text{rank}_6(9) \), which asks for the sum of the elements in the range from index 0 to 9. The tree is structured in such a way that each node represents a specific bit position of the binary representation of the elements in the array.

The table on the right side of the diagram shows the binary representation of the array elements, with the top row corresponding to the most significant bit and the bottom row to the least significant bit. Each column represents a bit position, with 1 indicating the presence of a 1 in that bit position and 0 indicating the absence.

By traversing the tree starting from the root and following the branches based on the query index, the tree efficiently calculates the range sum. In this case, the query \( \text{rank}_6(9) \) would involve following the path that includes the bits 1, 1, 0, 1, 1, 0, and 1, leading to the range sum being calculated as 110.
Wavelet Trees (2/2)

[0, 7]

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

[0, 3]

0 1 1 2 3
0 0 0 1 1

[4, 7]

6 7 5 4 6
1 1 0 0 1

rank₆(9)

1 1 0

[0, 1]

0 1 1
0 1 1

[2, 3]

2 3
0 1

[4, 5]

5 4
1 0

[6, 7]

6 7 6
0 1 0
Wavelet Trees (2/2)
Wavelet Trees (2/2)
Wavelet Trees (2/2)

Wavelet Trees are a data structure used for fast range queries on bit vectors. The diagram illustrates how wavelet trees are constructed and used. Each node represents a range of the bit vector, and the leaves represent the actual bit values.

The tree is constructed by dividing the range into smaller ranges, and each node corresponds to a range of the bit vector. The bit values are stored at the leaves of the tree, and the tree is traversed to answer range queries efficiently.

For example, to find the range [0, 7], you start at the root and traverse down the tree, selecting the path that corresponds to the range [0, 7]. The bit values at the leaves correspond to the bits in the range [0, 7].
The Intervals of a Wavelet Tree

- in each node, all represented characters share a bit prefix
- on depth $\ell$ the longest common bit prefix has length $\ell - 1$
- the bit prefixes form intervals
The Intervals of a Wavelet Tree

- in each node, all represented characters share a bit prefix
- on depth $\ell$ the longest common bit prefix has length $\ell - 1$
- the bit prefixes form intervals
The Intervals of a Wavelet Tree

- In each node, all represented characters share a bit prefix.
- On depth $\ell$, the longest common bit prefix has length $\ell - 1$.
- The bit prefixes form intervals.

- Finding characters in the wavelet tree requires finding the correct interval.
- Finding the position of a character requires finding the position in the last interval.

\[(\epsilon)_2\]
\[(0)_2\]
\[(1)_2\]
\[(00)_2\]
\[(01)_2\]
\[(10)_2\]
\[(11)_2\]
Rank-, Select-, and Access-Queries in Wavelet Trees (1/2)

**Rank-Queries**
- use rank queries on bit vectors
- at depth $\ell$ as for $\ell$-th MSB
- follow through tree according to bit
- as seen on a previous slide

Select-Queries
- identify leaf containing character
- select corresponding occurrence in leaf
- backtrack position up the tree to the root
- requires up and down traversal of the wavelet tree
  - see example on the board

Access-Queries
- follow bits through the wavelet tree
- return read bits
- same as rank but returning bit pattern instead of final rank
  - see example on the board
Rank-, Select-, and Access-Queries in Wavelet Trees (1/2)

**Rank-Queries**
- use rank queries on bit vectors
- at depth $\ell$ as for $\ell$-th MSB
- follow through tree according to bit
- as seen on a previous slide

**Select-Queries**
- identify leaf containing character
- select corresponding occurrence in leaf
- backtrack position up the tree to the root

**Access-Queries**
- follow bits through the wavelet tree
- return read bits
- same as rank but returning bit pattern instead of final rank

- requires up and down traversal of the wavelet tree
- see example on the board
Rank-, Select-, and Access-Queries in Wavelet Trees (1/2)

**Rank-Queries**
- use rank queries on bit vectors
- at depth $\ell$ as for $\ell$-th MSB
- follow through tree according to bit
- as seen on a previous slide

**Select-Queries**
- identify leaf containing character
- select corresponding occurrence in leaf
- backtrack position up the tree to the root
- requires up and down traversal of the wavelet tree
- see example on the board

**Access-Queries**
- follow bits through the wavelet tree
- return read bits
- same as rank but returning bit pattern instead of final rank
- see example on the board
Lemma: Query Times Wavelet Tree

Given a text $T$ over an alphabet of size $\sigma$, the wavelet tree of the text can answer rank, select, and access queries in $O(\lg \sigma)$ time.

Proof (Sketch)

All queries require
- just a constant number of rank and select queries on the bit vectors and
- at most one traversals from the root of the tree to a leaf and
- one traversal from a leaf to the root of the tree.
Bit Reversal Permutation

- given a bit representation of a character $\alpha$
- $reverse(\alpha)$ reverses the bits
- the MSB becomes the least significant bit

Definition: Bit-Reversal Permutation

The bit-reversal permutation $\rho_k$ is a permutation of the numbers $[0, 2^k)$ with

$$\rho_k(i) = reverse(i)$$

for $i \in [0, 2^k)$
given a bit representation of a character $\alpha$

$\text{reverse}(\alpha)$ reverses the bits

the MSB becomes the least significant bit

**Definition: Bit-Reversal Permutation**

The *bit-reversal permutation* $\rho_k$ is a permutation of the numbers $[0, 2^k)$ with

$$\rho_k(i) = \text{reverse}(i)$$

for $i \in [0, 2^k)$

- $\rho_2 = (0, 2, 1, 3) = ((00)_2, (10)_2, (01)_2, (11)_2)$
- $\rho_{k+1} = (2\rho_k(0), \ldots, 2\rho_k(2^k - 1), 2\rho_k(0) + 1, \ldots, 2\rho_k(2^k - 1) + 1)$
Bit Reversal Permutation

- given a bit representation of a character $\alpha$
- $reverse(\alpha)$ reverses the bits
- the MSB becomes the least significant bit

**Definition: Bit-Reversal Permutation**

The **bit-reversal permutation** $\rho_k$ is a permutation of the numbers $[0, 2^k)$ with

$$\rho_k(i) = reverse(i)$$

for $i \in [0, 2^k)$

- $\rho_2 = (0, 2, 1, 3) = ((00)_2, (10)_2, (01)_2, (11)_2)$
- $\rho_{k+1} = (2\rho_k(0), \ldots, 2\rho_k(2^k - 1), 2\rho_k(0) + 1, \ldots, 2\rho_k(2^k - 1) + 1)$

- same intervals as a wavelet tree
- used in the wavelet matrix
Alternative Representation

- alternative representation of wavelet trees
- removing tree structure
- only two areas per level (the intervals discussed before still exist)
Alternative Representation

- alternative representation of wavelet trees
- removing tree structure
- only two areas per level

Definition: Wavelet Matrix [CNP15]

Given a text $T$ of length $n$ over an alphabet of size $\sigma$, a wavelet matrix consists of

- bit vectors $BV_\ell$ for $\ell \in [1, \lceil \lg \sigma \rceil]$ of size $n$ and
- an array $Z[1..\sigma]$

Such that

- $Z[\ell]$ contains the number of zero bits in $BV_\ell$
- $BV_1$ contains all MSBs in text order
- $BV_\ell$ contains the $\ell$-th MSB the character at position $i$ in $BV_{\ell-1}$ at position
  - $rank_0(i)$ if $BV_{\ell-1} = 0$ and
  - $Z[\ell - 1] + rank_1(i)$ if $BV_{\ell-1} = 1$
Alternative Representation

- alternative representation of wavelet trees
- removing tree structure
- only two areas per level
- the intervals discussed before still exist

- better suited for large alphabets
- seemingly less structure
- retaining all important properties

**Definition: Wavelet Matrix [CNP15]**

Given a text $T$ of length $n$ over an alphabet of size $\sigma$ a wavelet matrix consists of

- bit vectors $BV_\ell$ for $\ell \in [1, \lceil \lg \sigma \rceil]$ of size $n$ and
- an array $Z[1..\sigma]$

Such that

- $Z[\ell]$ contains the number of zero bits in $BV_\ell$
- $BV_1$ contains all MSBs in text order
- $BV_\ell$ contains the $\ell$-th MSB the character at position $i$ in $BV_{\ell-1}$ at position
  - $\mathrm{rank}_0(i)$ if $BV_{\ell-1} = 0$ and
  - $Z[\ell - 1] + \mathrm{rank}_1(i)$ if $BV_{\ell-1} = 1$
Intervals of a Wavelet Matrix

- A wavelet matrix has the same intervals a wavelet tree has.
- Intervals not bounded by parent do no tree structure.

\[
\begin{array}{cccc}
(\epsilon)_2 \\
(0)_2 & (1)_2 \\
(00)_2 & (10)_2 & (01)_2 & (11)_2 \\
\end{array}
\]
**Intervals of a Wavelet Matrix**

- A wavelet matrix has the same intervals a wavelet tree has.
- Intervals not bounded by parent does not have a tree structure.

<table>
<thead>
<tr>
<th></th>
<th>ϵ&lt;sub&gt;2&lt;/sub&gt;</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(ε)</td>
<td></td>
<td>(0)</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>(00)</td>
<td></td>
<td>(10)</td>
<td>(01)</td>
<td>(11)</td>
</tr>
</tbody>
</table>

- Intervals of a wavelet tree (for comparison):
Example Wavelet Tree and Wavelet Matrix

<table>
<thead>
<tr>
<th>$BV_0$</th>
<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></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$BV_1$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>0</td>
<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>1</td>
</tr>
<tr>
<td>$BV_2$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></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>
<td>0</td>
</tr>
</tbody>
</table>


- queries on the wavelet matrix work similar
- example on the board 📝
Naive Wavelet Tree and Wavelet Matrix Construction (1/2)

Wavelet Tree

- first level are MSBs of characters of text
- for each level $\ell > 1$
  - stably sort text using Radix sort by bit prefixes of length $\ell - 1$
  - take $\ell$-th MSB of sorted sequence
  - sorted sequence is new text
Naive Wavelet Tree and Wavelet Matrix Construction (1/2)

Wavelet Tree
- first level are MSBs of characters of text
- for each level \( \ell > 1 \)
  - stably sort text using Radix sort by bit prefixes of length \( \ell - 1 \)
  - take \( \ell \)-th MSB of sorted sequence
  - sorted sequence is new text

Wavelet Matrix
- first level are MSBs of characters of text
- for each level \( \ell > 1 \)
  - stably sort text by \( \ell - 1 \) MSB
  - take \( \ell \)-th MSB of sorted sequence
  - sorted sequence is new text

![Wavelet Tree Diagram]

![Wavelet Matrix Diagram]
Lemma: Running Time and Memory Requirements Wavelet Tree and Wavelet Matrix

Given a text $T$ over an alphabet of size $\sigma$, the wavelet tree and wavelet matrix require $(1 + o(1))n\lceil \lg \sigma \rceil$ bits of space and can be constructed in $O(n \lg \sigma)$ time.
to make both fully functional bit vectors are augmented with binary rank and select support

Lemma: Running Time and Memory Requirements Wavelet Tree and Wavelet Matrix

Given a text $T$ over an alphabet of size $\sigma$, the wavelet tree and wavelet matrix require $(1 + o(1))n\lceil\lg \sigma\rceil$ bits of space and can be constructed in $O(n \lg \sigma)$ time

is there a asymptotically faster construction method
using requires broadword programming

every $\tau$-th level is a big level

big levels contain enough information to compute small levels below

small levels computed by splitting big levels

$O(b/\lg n)$ characters at a time with $b = o(\lg n)$

sketch on board 📚
Better Wavelet Tree Construction [Bab+15; MNV16]

- using requires broadword programming
- every $\tau$-th level is a big level
- big levels contain enough information to compute small levels below
- small levels computed by splitting big levels
- $O(b/\lg n)$ characters at a time with $b = o(\lg n)$
- sketch on board 📚

Lemma: Better Wavelet Tree Construction

Given a text $T$ over an alphabet of size $\sigma$, the wavelet tree and wavelet matrix require $(1 + o(1))n\lceil\lg \sigma\rceil$ bits of space and can be constructed in $O(n \lg \sigma / \sqrt{\lg n})$ time.
Better Wavelet Tree Construction [Bab+15; MNV16]

- using requires broadword programming
- every \( \tau \)-th level is a big level
- big levels contain enough information to compute small levels below
- small levels computed by splitting big levels
- \( O(b/ \log n) \) characters at a time with \( b = o(\log n) \)
- sketch on board 📚

**Lemma: Better Wavelet Tree Construction**

Given a text \( T \) over an alphabet of size \( \sigma \), the wavelet tree and wavelet matrix require \((1 + o(1))n\lceil \log \sigma \rceil \) bits of space and can be constructed in \( O(n \log \sigma / \sqrt{\log n}) \) time.

- can be implemented using AVX/SSE instructions [Kan18]
Huffman-shaped Wavelet Trees

- wavelet trees can be compressed
- more precise: the text can be compressed
- use Huffman codes
- wavelet trees cannot handle holes
- use canonical Huffman codes
Huffman-shaped Wavelet Trees

- wavelet trees can be compressed
- more precise: the text can be compressed
- use Huffman codes
- wavelet trees cannot handle holes
- use canonical Huffman codes

Huffman Codes (Recap)

- idea is to create a binary tree
- each character $\alpha$ is a leaf and has weight $\text{Hist}[^\alpha]$
- create node for two nodes without parent with smallest weight
- give new node total weight of children
- repeat until only one node without parent remains
- label edges:
  - left edge: 0
  - right edge: 1
- path to children gives code for character
Huffman-shaped Wavelet Trees

- wavelet trees can be compressed
- more precise: the text can be compressed
- use Huffman codes
- wavelet trees cannot handle holes
- use canonical Huffman codes

### Canonical Huffman Codes (Recap)

- start with Huffman codes, code word 0, and length 1
- to get canonical code for current length, then add 1 to code word
- to update length add 1 and append required amount of zeros to code word

### Huffman Codes (Recap)

- idea is to create a binary tree
- each character $\alpha$ is a leaf and has weight $\text{Hist}[^\alpha]$
- create node for two nodes without parent with smallest weight
- give new node total weight of children
- repeat until only one node without parent remains
- label edges:
  - left edge: 0
  - right edge: 1
- path to children gives code for character
Huffman-shaped Wavelet Trees

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$hc(\alpha)$</th>
<th>$chc(\alpha)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(11)$_2$</td>
<td>(11)$_2$</td>
</tr>
<tr>
<td>3</td>
<td>(01)$_2$</td>
<td>(10)$_2$</td>
</tr>
<tr>
<td>6</td>
<td>(100)$_2$</td>
<td>(011)$_2$</td>
</tr>
<tr>
<td>7</td>
<td>(101)$_2$</td>
<td>(010)$_2$</td>
</tr>
<tr>
<td>0</td>
<td>(0000)$_2$</td>
<td>(0011)$_2$</td>
</tr>
<tr>
<td>2</td>
<td>(0001)$_2$</td>
<td>(0010)$_2$</td>
</tr>
<tr>
<td>4</td>
<td>(0010)$_2$</td>
<td>(0001)$_2$</td>
</tr>
<tr>
<td>5</td>
<td>(0011)$_2$</td>
<td>(0000)$_2$</td>
</tr>
</tbody>
</table>

- Huffman codes (hc)
- canonical Huffman codes (chc) that are bit-wise negated
Huffman-shaped Wavelet Trees

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$hc(\alpha)$</th>
<th>$chc(\alpha)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$(11)_2$</td>
<td>$(11)_2$</td>
</tr>
<tr>
<td>3</td>
<td>$(01)_2$</td>
<td>$(10)_2$</td>
</tr>
<tr>
<td>6</td>
<td>$(100)_2$</td>
<td>$(011)_2$</td>
</tr>
<tr>
<td>7</td>
<td>$(101)_2$</td>
<td>$(010)_2$</td>
</tr>
<tr>
<td>0</td>
<td>$(0000)_2$</td>
<td>$(0011)_2$</td>
</tr>
<tr>
<td>2</td>
<td>$(0001)_2$</td>
<td>$(0010)_2$</td>
</tr>
<tr>
<td>4</td>
<td>$(0010)_2$</td>
<td>$(0001)_2$</td>
</tr>
<tr>
<td>5</td>
<td>$(0011)_2$</td>
<td>$(0000)_2$</td>
</tr>
</tbody>
</table>

- Huffman codes (hc)
- canonical Huffman codes (chc) that are bit-wise negated

- intervals are only missing to the right (white space)
- no holes allow for easy querying
Practical Sequential Wavelet Tree Construction

**Bottom-Up Construction [FKL18]**
- scan the text and create histogram
- while scanning compute first level
- use histogram to compute borders of intervals
- scan text again and fill bit vectors

- example on the next slide
Experimental Setup

- 64 GB RAM
- two Intel Xeon E5-2640v4 CPUs (10 cores at 2.4 GHz base frequency, 3.4 GHz maximum turbo frequency, and cache sizes: 32 KB L1D and L1I, 256 KB L2, 25.6 MB L3)

- same texts as in chapter 04
- results are average of 5 runs
Experiments: Sequential Wavelet Tree Construction

- Commoncrawl
- DNA
- Proteins
- Wikipedia

Throughput (Mbit/s) vs. input size $\lg n$ (B)

- naive
- pc
- pc.ss
- ps
- sdsl.pc
- serialWT
Parallel Wavelet Tree Construction in Practice

Domain Decomposition [Fue+17]
- create wavelet tree in parallel using $p$ PEs
- each PE gets a consecutive slice of text
- each PE builds partial wavelet tree for its text
- merge partial wavelet trees in parallel

- can utilize any sequential algorithm
- very fast in practice
- $O(n \lg \sigma / \sqrt{\lg n})$ work and $O(\sigma + \lg n)$ time
  [Shu20]
partial wavelet trees

compute wavelet tree

compute wavelet tree

compute wavelet tree

parallel merge

final wavelet tree
Experiments: Parallel Wavelet Tree Construction

![Graph showing throughput for different wavelet trees with varying PE counts]
Conclusion and Outlook

This Lecture
- wavelet tree and wavelet matrix
- Huffman-shaped wavelet trees
Conclusion and Outlook

This Lecture
- wavelet tree and wavelet matrix
- Huffman-shaped wavelet trees
- select on bit vectors
- practical algorithms for wavelet tree construction

Linear Time Construction
- ST
- SA
- WT
- LZ
- LCP
- BWT
Conclusion and Outlook

This Lecture
- wavelet tree and wavelet matrix
- Huffman-shaped wavelet trees
- select on bit vectors
- practical algorithms for wavelet tree construction

Next Lecture
- FM-index
- r-Index

Linear Time Construction

This diagram shows the relationships between different data structures:
- ST (Splay Tree)
- SA (Suffix Array)
- WT (Wavelet Tree)
- LZ (Lempel-Ziv)
- LCP (Longest Common Prefix)
- BWT (Burrows-Wheeler Transform)
Bibliography I


Bibliography II


