Text Indexing

Lecture 06: Burrows-Wheeler Transform

Florian Kurpicz
Recap: Text-Compression

Definition: LZ77 Factorization [ZL77]
Given a text $T$ of length $n$ over an alphabet $\Sigma$, the LZ77 factorization is
- a set of $z$ factors $f_1, f_2, \ldots, f_z \in \Sigma^+$, such that
- $T = f_1 f_2 \ldots f_z$ and for all $i \in [1, z]$ $f_i$ is
- single character not occurring in $f_1 \ldots f_{i-1}$ or
- longest substring occurring $\geq 2$ times in $f_1 \ldots f_i$

$T = \text{abababbbababa}\$
- $f_1 = a$
- $f_2 = b$
- $f_3 = \text{abal}$
- $f_4 = \text{bbb}$
- $f_5 = \text{aba}$
- $f_6 = \text{aba}$

Definition: LZ78 Factorization [ZL78]
Given a text $T$ of length $n$ over an alphabet $\Sigma$, the LZ78 factorization is
- a set of $z$ factors $f_1, f_2, \ldots, f_z \in \Sigma^+$, such that
- $T = f_1 f_2 \ldots f_z, f_0 = \epsilon$ and for all $i \in [1, z]$
- if $f_1 \ldots f_{i-1} = T[1..j-1]$, then $f_i$ is the longest prefix of $T[j..n]$, such that
  \[ \exists k \in [0, i), \alpha \in \Sigma \cup \{\}$ : $f_k = f_i\alpha \]

$T = \text{abababbbababa}\$
- $f_1 = a$
- $f_2 = b$
- $f_3 = ab$
- $f_4 = \text{abb}$
- $f_5 = \text{bb}$
- $f_6 = \text{aba}$
- $f_7 = \text{a}$
Definition: Burrows-Wheeler Transform

Given a text $T$ of length $n$ and its suffix array $SA$, for $i \in [1, n]$ the Burrows-Wheeler transform is

$$BWT[i] = \begin{cases} T[SA[i] - 1] & \text{for } SA[i] > 1 \\ $ & \text{for } SA[i] = 1 \end{cases}$$

- character before the suffix in $SA$-order
- choose characters cyclic $\$ for first suffix
- can compute $BWT$ in $O(n)$ time
- for binary alphabet $O(n/\sqrt{\log n})$ time and $O(n/\log n)$ words space is possible [KK19]

<table>
<thead>
<tr>
<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>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
</table>
| $T$ | a | b | a | b | c | a | b | c | a | b | b | a | $\
| $T$ | a | b | a | b | c | a | b | c | a | b | b | a | $\
| SA | 13 | 12 | 1 | 9 | 6 | 3 | 11 | 2 | 10 | 7 | 4 | 8 | 5 |
| LCP | 0 | 0 | 1 | 2 | 2 | 5 | 0 | 2 | 1 | 1 | 4 | 0 | 3 |
| BWT | a | b | $\$ | c | c | b | b | a | a | a | a | b | b |

- definition is not very descriptive
- easy way to compute $BWT$
- what can we do with the $BWT$
- PINGO can the BWT be reversed?
**Definition: Cyclic Rotation**

Given a text $T$ of length $n$, the $i$-th cyclic rotation is

$$T^{(i)} = T[i..n] \cdot T[1..i]$$

- $i$-th cyclic rotation is concatenation of $i$-th suffix and $(i-1)$-th prefix

**Definition: Burrows-Wheeler Transform (alt.)**

Given a text $T$ and a matrix containing all its cyclic rotations in lexicographical order as columns, then the **Burrows-Wheeler transform** of the text is the last row of the matrix.

$$T = \text{ababcabcabba}$$

<table>
<thead>
<tr>
<th>$T^{(1)}$</th>
<th>$T^{(2)}$</th>
<th>$T^{(3)}$</th>
<th>$T^{(4)}$</th>
<th>$T^{(5)}$</th>
<th>$T^{(6)}$</th>
<th>$T^{(7)}$</th>
<th>$T^{(8)}$</th>
<th>$T^{(9)}$</th>
<th>$T^{(10)}$</th>
<th>$T^{(11)}$</th>
<th>$T^{(12)}$</th>
<th>$T^{(13)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b a b c</td>
<td>a b a b c</td>
<td>a b a b c</td>
<td>a b a b c</td>
<td>a b a b c</td>
<td>a b a b c</td>
<td>a b a b c</td>
<td>a b a b c</td>
<td>a b a b c</td>
<td>a b a b c</td>
<td>a b a b c</td>
<td>a b a b c</td>
<td>a b a b c</td>
</tr>
<tr>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
</tr>
<tr>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
</tr>
<tr>
<td>c a b c a</td>
<td>c a b c a</td>
<td>c a b c a</td>
<td>c a b c a</td>
<td>c a b c a</td>
<td>c a b c a</td>
<td>c a b c a</td>
<td>c a b c a</td>
<td>c a b c a</td>
<td>c a b c a</td>
<td>c a b c a</td>
<td>c a b c a</td>
<td>c a b c a</td>
</tr>
<tr>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
</tr>
<tr>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
</tr>
<tr>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
</tr>
<tr>
<td>c a b a $</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
</tr>
<tr>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
</tr>
<tr>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
<td>b a b c a</td>
</tr>
<tr>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
<td>a $ a b a c</td>
</tr>
<tr>
<td>$ a b a c</td>
<td>$ a b a c</td>
<td>$ a b a c</td>
<td>$ a b a c</td>
<td>$ a b a c</td>
<td>$ a b a c</td>
<td>$ a b a c</td>
<td>$ a b a c</td>
<td>$ a b a c</td>
<td>$ a b a c</td>
<td>$ a b a c</td>
<td>$ a b a c</td>
<td>$ a b a c</td>
</tr>
</tbody>
</table>

$T = \text{ababcabcabba}$
First and Last Row

- two important rows in the matrix
- other rows are not needed at all
- there is a special relation between the two rows
  (later this lecture)

First Row $F$
- contains all characters or the text in sorted order

Last Row $L$
- is the BWT itself

$$T = \text{ababcabcabba}\$$(T_{13}T_{12}T_{11}T_{10}T_{9}T_{8}T_{7}T_{6}T_{5}T_{4}T_{3}T_{2}T_{1})$$

<table>
<thead>
<tr>
<th></th>
<th>$T_{13}$</th>
<th>$T_{12}$</th>
<th>$T_{11}$</th>
<th>$T_{10}$</th>
<th>$T_{9}$</th>
<th>$T_{8}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
</tr>
<tr>
<td></td>
<td>$\text{a}$</td>
<td>$\text{b}$</td>
<td>$\text{b}$</td>
<td>$\text{b}$</td>
<td>$\text{b}$</td>
<td>$\text{b}$</td>
</tr>
<tr>
<td></td>
<td>$\text{b}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
<td>$\text{b}$</td>
<td>$\text{c}$</td>
<td>$\text{a}$</td>
</tr>
<tr>
<td></td>
<td>$\text{a}$</td>
<td>$\text{b}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
</tr>
<tr>
<td></td>
<td>$\text{b}$</td>
<td>$\text{a}$</td>
<td>$\text{c}$</td>
<td>$\text{b}$</td>
<td>$\text{b}$</td>
<td>$\text{b}$</td>
</tr>
<tr>
<td></td>
<td>$\text{c}$</td>
<td>$\text{b}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
<td>$\text{b}$</td>
<td>$\text{a}$</td>
</tr>
<tr>
<td>$L$</td>
<td>$\text{a}$</td>
<td>$\text{b}$</td>
<td>$\text{b}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
</tr>
<tr>
<td></td>
<td>$\text{b}$</td>
<td>$\text{b}$</td>
<td>$\text{c}$</td>
<td>$\text{b}$</td>
<td>$\text{a}$</td>
<td>$\text{a}$</td>
</tr>
</tbody>
</table>
Properties of the BWT: Rank of Characters

**Definition: Rank**

Given a text $T$ over an alphabet $\Sigma$, the rank of a character at position $i \in [1, n]$ is

$$rank(i) = |\{j \in [1, i] : T[i] = T[j]\}|$$

- rank is number of same characters that occur before in the text
- mark ranks of characters w.r.t. text not BWT
- order of ranks is the same in first and last row

$T = ababcabcabba$

```
T     a b a b c a b c a b a $
rank  1 1 2 2 1 3 3 2 4 4 5 5 1
```
LF-Mapping (1/2)

- want to map characters from last to first row
- why do we want this?
  - helps with pattern matching
  - transform BWT back to \( T \)

**Definition: LF-mapping**

Given a text \( T \) of length \( n \) and its suffix array \( SA \), then the LF-mapping is a permutation of \([1, n]\), such that

\[
LF(i) = j \iff SA[j] = SA[i] - 1
\]

- similar to definition of BWT
- requires \( SA \) or explicitly saving LF-mapping

\[ T = ababcabcabba$ \]
**LF-Mapping (2/2)**

**Definition: C-Array and Rank-Function**

Given a text $T$ of length $n$ over an alphabet $\Sigma$, $\alpha \in \Sigma$, and $i \in [1, n]$ then

$$C[\alpha] = |i \in [1, n]: T[i] < \alpha|$$

and

$$\text{rank}_{\alpha}(i) = |\{j \in [1, i]: T[j] = \alpha\}|$$

- $C$ contains total number of smaller characters
- $\text{rank}_{\alpha}$ contains number of $\alpha$'s in prefix $T[1..i]$
- $\text{rank}_{\alpha}$ can be computed in $O(1)$ time [FM00]

![T a b a b c a b c a b b a $]

$rank$

| 1 | 1 | 2 | 2 | 1 | 3 | 3 | 2 | 4 | 4 | 5 | 5 | 1 |

- rank now on $BWT$
- $C$ is exclusive prefix sum over histogram

**Definition: LF-Mapping (alt.)**

Given a $BWT$, its $C$-array, and its $\text{rank}$-Function, then

$$LF(i) = C[BWT[i]] + \text{rank}_{BWT[i]}(i)$$
Reversing the BWT (1/2)

- characters (w.r.t. text) preserve order in L and F
- $LF$-mapping returns previous character in text

\[ T = \text{ababcabcabba}$\]
Reversing the BWT (2/2)

- characters (w.r.t. text) preserve order in $L$ and $F$
- $LF$-mapping returns previous character in text

- $T[n] = \$ \text{ and } T^{(n)}$ in first row
- apply $LF$-mapping on result to obtain any character

$$T[n - i] = L[LF(LF(\ldots(LF(1))\ldots))]$$

\[i - 1\text{ times}\]

<table>
<thead>
<tr>
<th>L</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>LF</td>
<td>a</td>
<td>b</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>
<td>b</td>
</tr>
</tbody>
</table>

\begin{align*}
T[13] &= \$ \text{ and } k = 1 \\
T[12] &= L[1] = a \text{ and } k = LF(1) = 2 \\
T[10] &= L[7] = b \text{ and } k = LF(7) = 9 \\
T[9] &= L[9] = a \text{ and } k = LF(9) = 4 \\
\ldots
\end{align*}
Properties of the BWT: Runs

- **BWT** is reversible
- can be used for lossless compression

**Definition: Run (simplified)**

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.)

**BWT** contains lots of runs
- same context is often grouped together 🍕
Compressing the BWT: Run-Length Compression

**Definition: Run-Length Encoding**
Given a text $T$, represent each run $T[i..i + \ell)$ as tuple

$(T[i], \ell)$

$T = \text{ababcabcabba}\$

<table>
<thead>
<tr>
<th>$i$</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>BWT</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>b</td>
<td>b</td>
</tr>
</tbody>
</table>

- $(a, 1)$
- $(b, 1)$
- $(\$, 1)$
- $(c, 2)$
- $(b, 2)$
- $(a, 4)$
- $(b, 2)$
Compressing the BWT: Move-to-Front

**Definition: Move-To-Front Encoding**

Given a text $T$ over an alphabet $\Sigma = [1, \sigma]$, the MTF encoding $MTF(T)$ of the text is computed as follows:

- start with a list $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$.
- scan text from left to right, for character $T[i]$
  - append position of $T[i]$ in $X$ to $MTF(T)$ and
  - move $T[i]$ to front of $X$.

- MTF encoding can easily be reverted.
- consists of many small numbers.
- runs are preserved.
- use Huffman on encoding.

**Example**

Let $T = \text{ababcabcabba}$.

<table>
<thead>
<tr>
<th>$T$</th>
<th>1 2 3 4 5 6 7 8 9 0 11 12 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWT</td>
<td>a b $ c c b b a a a b b</td>
</tr>
</tbody>
</table>

- $X = $, a, b, c
- $MTF = 2$ and $X = a, $, b, c
- $MTF = 23$ and $X = b, a, $, c
- $MTF = 233$ and $X = $, b, a, c
- $MTF = 2334$ and $X = c, $, b, a
- $MTF = 23341$ and $X = c, $, b, a
- $MTF = 233411$ and $X = c, $, b, a
- $MTF = 23341131411121$
Pattern Matching using the BWT

**Recap**

Given a text $T$ of length $n$ over an alphabet $\Sigma$, $\alpha \in \Sigma$, and $i \in [1, n]$ then

$$C[\alpha] = |i \in [1, n]: T[i] < \alpha|$$

and

$$rank_\alpha(i) = |\{j \in [1, i]: T[j] = \alpha\}|$$

- interval for $\alpha$ is $[C[\alpha - 1], C[\alpha + 1]]$
- find sub-interval using $rank_\alpha$
- find interval of occurrences in $SA$ using $BWT$
- $SA$ is divided into intervals based on first character of suffix as seen during $SAIS$
- text from $BWT$ is backwards
- search pattern backwards
Backwards Search in the BWT

Function BackwardsSearch(P[1..n], C, rank):
1  
2  
3  
4  
5  
6  
7  

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
Sampling the Suffix Array

- reporting queries require SA
- storing whole SA requires too much space
- better: sample every s-th SA position in SA'

how to find sampled position?
- mark sampled positions in bit vector of size n
- if match occurs check if position is sampled
- otherwise find sample using LF
- if SA[i] = j then SA[LF(i)] = j − 1

- rank_1(i) in bit vector is number of sample
- SA'[rank_1(i)] is sampled value
- SA'[rank_1(i)] + steps till sample found is correct SA value

- finding a sample requires O(s · t_{rank}) time
Efficient Bit Vectors in Practice (1/3)

**std::vector<char/int/...>**
- easy access
- very big: 1, 4, ... bytes per bit

**std::vector<bool>**
- bit vector in C++ (1 bit per byte)
- easy access
- layout depending on implementation

**std::vector<uint64_t>**
- requires 8 bytes per bit(?)
- store 64 bits in 8 bytes
- how to access bits

\[
i/64 \text{ is position in 64-bit word}
\]
\[
i \% 64 \text{ is position in word}
\]
// There is a bit vector
std::vector<uint64_t> bit_vector;

// access i-th bit
uint64_t block = bit_vector[i/64];
bool bit = (block >> (63 - (i % 64))) & 1ULL;
Efficient Bit Vectors in Practice (3/3)

- \((\text{block} \gg (63-(i\%64))) \& 1\text{ULL};\)
  - Fill bit vector from left to right
    - \begin{array}{ccccccccc}
        0 & 1 & 2 & 3 & 4 & 5 & \ldots & 62 & 63 \\
        1 & 1 & 1 & 0 & 1 & 0 & \ldots & 1 & 0 \\
    \end{array}
    - Assembler code:
      - `mov ecx, edi`
      - `not ecx`
      - `shr rsi, cl`
      - `mov eax, esi`
      - `and eax, 1`

- \((\text{block} \gg (i\%64)) \& 1\text{ULL};\)
  - Fill bit vector right to left
    - \begin{array}{ccccccccc}
        63 & 62 & \ldots & 5 & 4 & 3 & 2 & 1 & 0 \\
        0 & 1 & \ldots & 0 & 1 & 0 & 1 & 1 & 1 \\
    \end{array}
    - Assembler code:
      - `mov ecx, edi`
      - `shr rsi, cl`
      - `mov eax, esi`
      - `and eax, 1`
Rank Queries in Bit Vectors (1/2)

\[ \text{rank}_\alpha(i) \] # of \( \alpha \)s before \( i \)

\[ \text{select}_\alpha(j) \] position of \( j \)-th \( \alpha \)

block

super-block

\[ \text{rank}_0(5) \]

\[ \text{select}_1(5) \]

# of 0s w.r.t. super-block

# of 0s w.r.t. BV
Rank Queries in Bit Vectors (2/2)

- for a bit vector of size $n$
  - blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
  - super blocks of size $s' = s^2 = \Theta(\lg^2 n)$

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

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

PINGO how fast can rank queries be answered?

- query in $O(1)$ time
  - $\text{rank}_0(i) = i - \text{rank}_1(i)$
The FM-Index (First Look) [FM00]

Building Blocks of FM-Index
- wavelet tree on BWT providing rank-function
- wavelet trees are topic of next lecture!
- 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 Space Requirements
Given a text $T$ of length $n$ over an alphabet of size $\sigma$, the FM-index requires $O(n \lg \sigma)$ bits of space

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

This Lecture
- Burrows-Wheeler transform
- introduction to FM-index
- efficient bit vectors
- rank queries on bit vectors

Next Lecture
- wavelet trees
- more on FM-index

Linear Time Construction
Bibliography I


