Text Indexing: Lecture 2

Simon Gog – gog@kit.edu
### The Burrows-Wheeler Transform

We have already defined the Burrows-Wheeler transformed string $T_{BWT}$ in Section 2.3. In this section, we will describe the relation of $T_{BWT}$ with the LF and Œ function.

Before doing this, we first have a look at the interesting history of the Burrows-Wheeler transform. David Wheeler had the idea of the character reordering already in 1978. It then took 16 years and a collaboration with Michael Burrows until the seminal technical report at Digital Equipment Corporation [BW94] was published. The reader is referred to [ABM08] for the historical background of this interesting story. Today, the Burrows-Wheeler Transform plays a key role in text data compression. The most prominent example for a compressor based on this transformation is the bzip2 application.

However, one can not only compress the text but also index it. To show how this is possible, we have to present the relation between the LF and Œ function and $T_{BWT}$.

---

#### Figure 2.5 (a) and (b)

![Figure 2.5](image.png)

**Figure 2.5:** The arrows depict the (a) LF function and (b) the Œ function and how it can be expressed by $T_{BWT}$ and $F$ for the running example $T = \text{umulmundumulmum}$.
Self Index
Does not only provide search functionality but also efficient reconstruction of any substring of the original text.

LF mapping
For every suffix \( j = SA[i] \), \( LF(i) \) is the position of \( j - 1 \) (the previous suffix in the text) in \( SA \). It holds:

\[
LF[i] = C[BWT[i]] + \text{rank}(i, BWT[i], BWT)
\]

I.e. we can decode text backwards. Starting from the last suffix $ at SA-position 0, we can decode the whole text.
Turning the FM-Index into a Self Index

Inverse Suffix Array

<table>
<thead>
<tr>
<th>i</th>
<th>SA</th>
<th>ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

- Inverse permutation of SA: $ISA[SA[i]] = i$
- Given suffix $x$. Where does $x$ occur in $SA$?

Express $LF$:  
$$LF[i] = ISA[SA[i] - 1 \mod n]$$

Express $\Psi$:  
$$\Psi[i] = ISA[SA[i] + 1 \mod n]$$
Implement $\Psi$ via WT over BWT

$\Psi$ calculation
$\Psi[i] = \text{select}(\text{rank}(i, F[i], F), F[i], \text{BWT})$

Operation select
Given a sequence $X$, a symbol $c$, and an integer $i$. Operation $\text{select}(i, c, X)$ returns the position of the $i$-th occurrence of $c$ in $X$.

Exercise
- Assume that there is a data structure which solves select queries on bitvectors in constant time using $o(n)$ space. Show how select can be implemented in $\log \sigma$ time and $o(n \log \sigma)$ bits for a sequence of length $n$ over an alphabet of size $\sigma$.
- What is the maximal size of the set $\{ i \mid \Psi[i] > \Psi[i + 1] \}$?
Sampling (for locate)

Fix a sampling rate $s$. Add a bitvector $B$ of length $n$ with $B[i] = 1$ if $SA[i] \equiv 0 \mod s$. Store the samples in array $SA'$ of size $n/s$. I.e. for all $i$ with $B[i] = 1$, $SA'[\text{rank}(i, 1, B)] = SA[i]$.

Pseudo-code for accessing $SA[i]$

See blackboard.
Compressing the Index

Definitions

$H_0(X)$ – zeroth order empirical entropy

Given a sequence $X$ of length $n$ over alphabet $\Sigma$. Let $n_c$ be the number of occurrences of $c \in \Sigma$ in $X$.

$$H_0(X) = \sum_{c \in \Sigma, n_c > 0} \frac{n_c}{n} \log \frac{n}{n_c}$$

Provides a lower bound to the number of bits needed to compress $X$ using a compressor which just considers character frequencies.
Elias-Fano Coding [1, 2]

Given a non-decreasing sequence $X$ of length $m$ over alphabet $[0..n]$. $X$ can be represented using $2m + m \log \frac{n}{m} + o(m)$ bits while each element can still be accessed in constant time.

This representation can also be used to represent a bitvector (e.g. $n$ is bitvector length, $m$ the number of set bits, and $X$ the position of the set bits)
Compressing the Index
How does Elias-Fano coding work?

- Divide each element into two parts: high-part and low-part.
- \(\lfloor \log m \rfloor\) high-bits and \(\lceil \log n \rceil - \lfloor \log m \rfloor\) low bits
- Sequence of high-parts of \(X\) is also non-decreasing.
- Gap encode the high-parts and use unary encoding to represent gaps. Call result \(H\).
- I.e. for a gap of size \(g_i\) we use \(g_i + 1\) bits (\(g_i\) zeros, 1 one).
- Sum of gaps (= \#zeros) is at most \(2^{\lfloor \log m \rfloor} \leq 2^{\log m} = m\)
- I.e. \(H\) has size at most \(2m\) (\#zeros + \#ones)
- Low-parts are represented explicitly.
How does Elias-Fano coding work?

\[ X = \begin{array}{cccccccc}
4 & 13 & 15 & 24 & 26 & 27 & 29 \\
00100 & 01101 & 01111 & 11000 & 11010 & 11011 & 11101 \\
0 & 4 & 1 & 5 & 1 & 7 & 3 & 0 & 3 & 2 & 3 & 3 & 3 & 5 \\
0-0 & 1-0 & 1-1 & 3-1 & 3-3 & 3-3 & 3-3 & 3-3 \\
\delta = 0 & 1 & 0 & 2 & 0 & 0 & 0 & 0 \\
H = 1011001111 \\
L = 4 & 5 & 7 & 0 & 2 & 3 & 5
\end{array}\]
Compressing the Index
How does Elias-Fano coding work?

Constant time access

- Add a select structure to $H$ ([4]).

00  $access(i)$
01  $p \leftarrow select(i + 1, 1, H)$
02  $x \leftarrow p - i$
03  $return \ x \cdot 2^{\lfloor \log n \rfloor - \lfloor \log m \rfloor + L[i]}$
Compressing the Index
Apply Elias-Fano coding to a $\Psi$-based CSA

- $\Psi$ consists of at most $\sigma$ non-decreasing sequences in the range $[0, n - 1]$.

$$|\text{CSA}_\Psi| = \sum_{c \in \Sigma} \left( n_c (2 + \log \frac{n}{n_c}) + o(n_c) \right)$$

$$= \sum_{c \in \Sigma} 2n_c + n \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c} + o(n)$$

$$= 2n + nH_0(T) + o(n)$$

+ $O(\sigma \log n)$ bits to handle character boundaries
Compressing the Index

Search in a $\Psi$-based CSA

- Compare pattern from left to right (forward search) to suffix $SA[i]$
- Use binary search on the interval $[0, n - 1]$.

00 compare($P$, $i$)
01 $k \leftarrow 0$
02 while $k < |P|$ do
03     if $C[P[k] + 1] - 1 < i$ then
04         return $-1$  // $P$ smaller than suffix
05     else if $C[P[k]] > i$ then
06         return $+1$  // $P$ larger than suffix
07         $k \leftarrow k + 1$
08         $i \leftarrow \Psi[i]$
09 return $0$    // $P$ equal to the first $m$ character of the suffix
Compressing the Index
Using self-delimiting codes

E.g. Elias-δ code. Let $bin(x)$ be the binary representation of $x$. Write $\lfloor bin(\lfloor bin(x) \rfloor) \rfloor - 1$ in unary, append the $\lfloor bin(\lfloor bin(x) \rfloor) \rfloor - 1$ least significant bits of $\lfloor bin(x) \rfloor$, and append the $\lfloor bin(x) \rfloor - 1$ least significant bits of $bin(x)$.

| $x_{(10)}$ | $x_{(\text{unary})}$ | $x_{(2)}$ | $x_{(\delta\text{-code})}$ | $|x_{\delta\text{-code}}|$ |
|---|---|---|---|---|
| 1 | 01 | 1 | 1 | 1 |
| 2 | 001 | 10 | 0100 | 4 |
| 3 | 0001 | 11 | 0101 | 4 |
| 4 | 00001 | 100 | 01100 | 5 |
| 5 | 000001 | 101 | 01101 | 5 |
| 13 | 000000000000001 | 1101 | 00100101 | 8 |

Length of Elias-δ code for $x$ is $2 \log \log x + \log x + O(1)$ bits.
Compressing the Index
Space analysis of a $\Psi$-based CSA using Elias-$\delta$ code.

For each character $c$ gap-encode its increasing $\Psi$ sequence. E.g.
$g_{c,i} = \Psi[C[c] + i] - \Psi[C[c] + i - 1]$ for $i > 0$ and $g_{c,0} = \Psi[C[c]]$ for $i = 0$.

$$\sum_{c \in \Sigma} \sum_{i=0}^{n_c - 1} (\log g_{c,i} + 2 \log \log g_{c,i} + O(1))$$

$$\leq O(n) + \sum_{c \in \Sigma} \sum_{i=0}^{n_c - 1} \left( \log \frac{n}{n_c} + 2 \log \log \frac{n}{n_c} \right)$$

$$= O(n) + n \sum_{c \in \Sigma} \frac{n_c}{n} \left( \log \frac{n}{n_c} + 2 \log \log \frac{n}{n_c} \right)$$

$$= nH_0(T) + O(n \log \log n)$$
Another approach to compress the index is to use *compressed bitvectors* for the wavelet tree instead of a plain bitvector (`bit_vector`). There are two basic compressed bitvector representations:

- Elias-Fano coded bitvector (`sd_vector`); see Okanohara & Sadakane [4]
- $H_0$-compressed bitvector (`rrr_vector`); see Raman et al. [5]
Let $B$ be a bitvector of length $n$ and $\kappa$ be the number of set bits.
- Let $X$ be the sorted list of positions of the set bits in $B$.
- Apply Elias-Fano coding on $X$.
- Space: $2\kappa + \kappa \log \frac{n}{\kappa} + o(\kappa)$
- $t_{\text{select}} \in O(1)$
- $t_{\text{access}} \in O(\log \kappa)$
- $t_{\text{rank}} \in O(\log \kappa)$
Let $B$ be a bitvector of length $n$.

$$H_0(B) = \frac{\kappa}{n} \log \frac{n}{\kappa} + \frac{n - \kappa}{n} \log \frac{n}{n - \kappa},$$

where $\kappa = \# \text{ of set bits in } B$.

**Theorem (Raman et al. [5])**

A bitvector can be represented in $nH_0(B) + o(n)$ bits of space. At the same time rank queries can be performed in constant time.
$H_0$-compressed bitvector

- Split $B$ into block of $K = \frac{1}{2} \log n$ bits
- For each block store the number of set bits (in $\lceil \log K + 1 \rceil$ bits)
- In total these class identifiers sum up to $O(n \frac{\log \log n}{\log n})$ bits

- Represent a block as tuple $(\kappa_i, r_i)$, $0 \leq \kappa_i \leq K$ is the class identified and the index $r_i$ within class $\kappa_i$. $r_i \in [0, \binom{K}{\kappa_i} - 1]$.
- The class indexes sum up to

$$\left\lceil \log \binom{K}{\kappa_0} \right\rceil + \cdots + \left\lceil \log \binom{K}{\kappa_{(n-1)/K}} \right\rceil \leq \log \left( \binom{K}{\kappa_0} \times \cdots \times \binom{K}{\kappa_{(n-1)/K}} \right) + n/K$$

$$\leq \log \left( \frac{n}{\kappa_0 + \cdots + \kappa_{(n-1)/K}} \right) + n/K = \log \binom{n}{\kappa} + n/K = nH_0(B) + O(n/\log n)$$
$H_0$-compressed bitvector

- Lookup table to map between class indexes and block
- Overall space: $nH_0(B) + O\left(\frac{n}{\log n}\right) + O(n\frac{\log \log n}{\log n}) = nH_0(B) + o(n)$
- Rank structure: Absolute rank samples + relative rank samples + lookup tables for blocks of size $K = \frac{1}{2} \log n$.
- Note: Four-Russian trick again
- Problems in practice:
  - Lookup tables should fit in cache; therefore $K \approx 15$
  - For $K = 15$ class identifiers are not negligible
\textit{H}_{0}\text{-compressed bitvector}

<table>
<thead>
<tr>
<th>Block size (K)</th>
<th>Space in (%) of original bitvector</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>120</td>
</tr>
<tr>
<td>32</td>
<td>100</td>
</tr>
<tr>
<td>64</td>
<td>80</td>
</tr>
<tr>
<td>128</td>
<td>60</td>
</tr>
<tr>
<td>256</td>
<td>40</td>
</tr>
</tbody>
</table>

![Graph showing space consumption for different block sizes and bitvectors](image)
On-the-fly block en/de-coding

- Use combinatorial number system of degree $\kappa_i$ to en/de-code a block (Navarro & Providel [3])
- Greedy algorithm is used to en/de-code block
- Required operations:
  - comparison
  - addition/subtraction
On-the-fly block en/de-coding

Figure 5: Encoding of block 100101 into the 5-bit number 13 (5 = d log_6 3 e).

Figure 6: Decoding of tuple (n = 6, bt[2] = 3, btnr[2] = 13) to block 100101.
On-the-fly block en/de-coding

Figure 5: Encoding of block 100101 into the 5-bit number 13 ($5 = \log_6 3^e$).

0 ≥ 0 = \binom{0}{1}
0 < 1 = \binom{1}{1}
0 < 2 = \binom{2}{1}
3 ≥ 3 = \binom{3}{2}
3 < 6 = \binom{4}{2}
13 ≥ 10 = \binom{5}{3}
$H_0$-compressed bitvector

![Graph showing time per operation in μs vs block size K for different bitvectors.]

**Legend:**
- SEL-$R^3K$
- RANK-$R^3K$
- BV-$R^3K$ access

**Bitvectors:**
- bitvector = WEB-WT-1GB
- bitvector = DNA-WT-1GB

*Figure 6. Query times for the operations on the $H_0$-compressed bitvector as function of block size $K$. The sample rate $t$ was set to 32.*
$H_0$-compression for sequences

Let $S$ be a sequence of length $n$ over alphabet $\Sigma$ of size $\sigma$.

$$H_0(S) = \sum_{c \in [0,\sigma-1]} \frac{n_c}{n} \log \frac{n}{n_c}$$

where $n_c$ is the number of occurrences of symbol $c$ in $S$.

**Idea**

Represent $S$ as a wavelet tree and use $H_0$-compressed bitvectors
**Notation**

A wavelet tree $WT(S)$ of a sequence $S[0, n-1]$ over an alphabet $\Sigma[0, \sigma-1]$ is defined as a perfectly balanced binary tree of height $H = \lceil \log \sigma \rceil$. Conceptually the root node $v_\epsilon$ represents the whole sequence $S_{v_\epsilon} = S$. The left (right) child of the root represents the subsequence $S_0$ ($S_1$) which is formed by only considering symbols of $X$ which are prefixed by a 0-bit (1-bit). In general the $i$-th node on level $L$ represents the subsequence $X_{i(2)}$ of $X$ which consists of all symbols which are prefixed by the length $L$ binary string $i(2)$. More precisely the symbols in the range $R(v_{i(2)}) = [i \cdot 2^{H-L}, (i+1) \cdot 2^{H-L} - 1]$. Let $n_{i(2)}$ be the size of $v_{i(2)}$ and $B_{i(2)}$ the bitvector which consists of the $\ell$-th bits of $S_{i(2)}$. 
$H_0$-compression for sequences

Let $\omega$ be a prefix of a binary string of length $L - 1$. Assume that the space to represent subsequences $S_{\omega_0}$ and $S_{\omega_1}$ using a WT is $n_{\omega_0}H_0(S_{\omega_0})$ and $n_{\omega_1}H_0(S_{\omega_0})$. The space to represent $S_{\omega}$ is

\[
= n_\omega H_0(B_\omega) + n_{\omega_0}H_0(S_{\omega_0}) + n_{\omega_1}H_0(S_{\omega_1})
\]

\[
= n_{\omega_0} \cdot \log \frac{n_\omega}{n_{\omega_0}} + n_{\omega_1} \cdot \log \frac{n_\omega}{n_{\omega_1}} + n_{\omega_0}H_0(S_{\omega_0}) + n_{\omega_1}H_0(S_{\omega_1})
\]

\[
= n_{\omega_0} \cdot \log \frac{n_\omega}{n_{\omega_0}} + n_{\omega_0}H_0(S_{\omega_0}) + n_{\omega_1} \cdot \log \frac{n_\omega}{n_{\omega_1}} + n_{\omega_1}H_0(S_{\omega_1})
\]

For (a) we get with the definition of $n_{\omega_0}H_0(S_{\omega_0}) = \sum_{\alpha \in \sigma^{H-L}} n_{\omega_0\alpha} \log \frac{n_{\omega_0}}{n_{\omega_0\alpha}}$

\[
(a) = \sum_{\alpha \in \sigma^{H-L}} n_{\omega_0\alpha} \left( \log \frac{n_\omega}{n_{\omega_0}} + \log \frac{n_{\omega_0}}{n_{\omega_0\alpha}} \right) = \sum_{\alpha \in \sigma^{H-L}} n_{\omega_0\alpha} \log \frac{n_\omega}{n_{\omega_0\alpha}}
\]
$H_0$-compression for sequences

Space to represent $S_\omega$ by adding (a) and (b)

$$
= \sum_{\alpha \in \sigma^{H-L}} n_{\omega \alpha} \log \frac{n_\omega}{n_{\omega 0 \alpha}} + \sum_{\alpha \in \sigma^{H-L}} n_{\omega 1 \alpha} \log \frac{n_\omega}{n_{\omega 1 \alpha}}
$$

$$
= \sum_{\alpha' \in \sigma^{H-L-1}} n_{\omega \alpha'} \log \frac{n_\omega}{n_{\omega \alpha'}}
$$

$$
= n_\omega H_0(S_\omega)
$$

Induction start for $L = H$ (leaf nodes of WT). For a single symbol $\omega' \in \Sigma$ we get $H_0(S_{\omega'}) = 0$. 
Higher order empirical entropy

Let $C$ be the set of all (distinct) substrings of length $k$ in $T$. For a fixed context $c \in C$ we define $T_c$ to be the concatenation of all characters which follow $c$ in $S$. Then the $k$th order entropy is defined as

$$H_k(T) = \sum_{c \in C} \frac{|S_c|}{n} H_0(S_c)$$

Example $T = \text{ananas}, k = 2$

$C = \{\text{an}, \text{na}, \text{as}\}$

$S_{\text{an}} = \text{aa}, S_{\text{na}} = \text{ns}, S_{\text{as}} = \epsilon$

$\rightarrow H_2(T) = \frac{2}{6} H_0(\text{ns}) = \frac{1}{3}$ bits
### $H_k$ of Pizza&Chili corpus texts (200MB versions)

<table>
<thead>
<tr>
<th>$k$</th>
<th>DBLP.XML</th>
<th>DNA</th>
<th>ENGLISH</th>
<th>PROTEINS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.26</td>
<td>1.97</td>
<td>4.53</td>
<td>4.20</td>
</tr>
<tr>
<td>1</td>
<td>3.48</td>
<td>1.93</td>
<td>3.62</td>
<td>4.18</td>
</tr>
<tr>
<td>2</td>
<td>2.17</td>
<td>1.92</td>
<td>2.95</td>
<td>4.16</td>
</tr>
<tr>
<td>3</td>
<td>1.43</td>
<td>1.92</td>
<td>2.42</td>
<td>4.07</td>
</tr>
<tr>
<td>4</td>
<td>1.05</td>
<td>1.91</td>
<td>2.06</td>
<td>3.83</td>
</tr>
<tr>
<td>5</td>
<td>0.82</td>
<td>1.90</td>
<td>1.84</td>
<td>3.16</td>
</tr>
<tr>
<td>6</td>
<td>0.70</td>
<td>1.88</td>
<td>1.67</td>
<td>1.50</td>
</tr>
</tbody>
</table>

$H_k(T)$ in bits contexts/| $T$| in percent
Question: How can we adjust the FM-index to use just $H_k(T) + O(\sigma^k)$ bits of space?
Bibliography


