Text Indexing: Lecture 2

Simon Gog – gog@kit.edu
2.5 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 $\psi$ and $\phi$ 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 $\psi$ and $\phi$ function and $T_{BWT}$.

Figure 2.5 (a) shows once again $\phi$, $T_{BWT}$ and the sorted suffixes of $T$. Now remember that the $\phi$ function at position $i$ tells us for suffix $SA[i]$ the previous suffix in the text, i.e. the position of suffix $SA[i] \neq 1$. We take for example suffix $SA[4] = 14$ which spells out $m$. Now, as $T_{BWT}[4] = u$, we know that suffix $SA[13]$ starts with character $u$. I.e.

![Diagram](image-url)
Turning the FM-Index into a Self Index

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:

$$LF[i] = C[BWT[i]] + rank(i, BWT[i], BWT)$$

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 $</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>6 dumulmum$</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>11 lmum$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3 lmundumulmum$</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>8 m$</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>15 mulmum$</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>9 mulmundumulmum$</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>1 mum$</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>13 mundumulmum$</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>5 ndumulmum$</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10 ulmum$</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>2 ulmundumulmum$</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>7 um$</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>12 umulmum$</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>4 umulmundumulmum$</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>0 undumulmum$</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

$\mathcal{H}_0(X)$ – 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 $X$.

$$\mathcal{H}_0(X) = \sum_{c \in \Sigma, n_c > 0} \frac{n_c}{n} \log \frac{n}{n_c}$$

Provides a lower bound to the number of bits needed to compress $X$ using a compressor which just considers character frequencies.
Compressing the Index

Definitions

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 ($= \#\text{zeros}$) is at most $2^{\lfloor \log m \rfloor} \leq 2^{\log m} = m$
- I.e. $H$ has size at most $2m (\#\text{zeros} + \#\text{ones})$
- Low-parts are represented explicitly.
Compressing the Index

How does Elias-Fano coding work?

\[ X = \begin{array}{cccccccc}
4 & 13 & 15 & 24 & 26 & 27 & 29 \\
00100 & 01101 & 01111 & 11000 & 11010 & 11011 & 11101 \\
\end{array} \]

\[ \delta = \begin{array}{cccccccc}
0 & 1 & 0 & 2 & 0 & 0 & 0 \\
0-0 & 1-0 & 1-1 & 3-1 & 3-3 & 3-3 & 3-3 \\
\end{array} \]

\[ H = \begin{array}{ccccc}
1011001111 \\
\end{array} \]

\[ L = \begin{array}{ccccccc}
4 & 5 & 7 & 0 & 2 & 3 & 5 \\
\end{array} \]
How does Elias-Fano coding work?

Constant time access

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

```plaintext
00 access(i)
01 p ← select(i + 1, 1, H)
02 x ← 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 \hspace{1em} \textbf{compare}(P, i) \\
01 \hspace{1em} k \leftarrow 0 \\
02 \hspace{1em} \textbf{while } k < |P| \hspace{1em} \textbf{do} \\
03 \hspace{2em} \textbf{if } C[P[k] + 1] - 1 < i \hspace{1em} \text{then} \\
04 \hspace{3em} \textbf{return } -1 \hspace{1em} \text{//P smaller than suffix} \\
05 \hspace{2em} \textbf{else if } C[P[k]] > i \hspace{1em} \text{then} \\
06 \hspace{3em} \textbf{return } +1 \hspace{1em} \text{//P larger than suffix} \\
07 \hspace{2em} k \leftarrow k + 1 \\
08 \hspace{2em} i \leftarrow \Psi[i] \\
09 \hspace{1em} \textbf{return } 0 \hspace{1em} \text{//P equal to the first } m \text{ character of the suffix}
Compressing the Index
Using self-delimiting codes

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

\[
\begin{array}{cccc}
X_{(10)} & X_{(\text{unary})} & X_{(2)} & X_{(\delta-\text{code})} & |X_{\delta-\text{code}}| \\
1 & 01 & 1 & 1 & 1 \\
2 & 001 & 10 & 0100 & 4 \\
3 & 0001 & 11 & 0101 & 4 \\
4 & 00001 & 100 & 01100 & 5 \\
5 & 000001 & 101 & 01101 & 5 \\
13 & 000000000000001 & 1101 & 00100101 & 8 \\
\end{array}
\]

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 $g_{c,i} = \Psi[C[c]]$ for $i = 0$.

\[
\sum_{c \in \Sigma} \sum_{i=0}^{n_c-1} (\log g_{c,i} + 2 \log \log g_{c,i} + O(1)) 
\leq \quad O(n) + \sum_{c \in \Sigma} \sum_{i=0}^{n_c-1} \left( \log \frac{n}{n_c} + 2 \log \log \frac{n}{n_c} \right) 
= \quad O(n) + n \sum_{c \in \Sigma} \frac{n_c}{n} \left( \log \frac{n}{n_c} + 2 \log \log \frac{n}{n_c} \right) 
= \quad n\mathcal{H}_0(T) + O(n \log \log n) 
\]
Another approach to compress the index is to use *compressed bitvectors* for the wavelet tree instead of a plain bitvector (\textit{bit\_vector}). There are two basic compressed bitvector representations:

- Elias-Fano coded bitvector (sd\_vector); see Okanohara & Sadakane [4]
- $\mathcal{H}_0$-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: $2\kappa + \kappa \log \frac{n}{\kappa} + o(\kappa)$
- $t_{select} \in O(1)$
- $t_{access} \in O(\log \kappa)$
- $t_{rank} \in O(\log \kappa)$
$H_0$-compressed bitvector

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.
**$\mathcal{H}_0$-compressed bitvector**

- Split $B$ into blocks of $K \approx \frac{1}{2} \log n$ 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

\[
\left\lfloor \log \left( \binom{K}{\kappa_0} \right) \right\rfloor + \cdots + \left\lfloor \log \left( \binom{K}{\kappa_{(n-1)}/K} \right) \right\rfloor \leq \log \left( \binom{K}{\kappa_0} \right) \times \cdots \times \binom{K}{\kappa_{(n-1)}/K} + n/K
\]

\[
\leq \log \left( \binom{n}{\kappa_0 + \cdots + \kappa_{(n-1)}/K} \right) + n/K = \log \left( \binom{n}{\kappa} \right) + n/K = n\mathcal{H}_0(B) + O(n/\log n)
\]
**H₀-compressed bitvector**

- Lookup table to map between class indexes and block
- Overall space: \( nH₀(B) + \mathcal{O}\left(\frac{n}{\log n}\right) + \mathcal{O}\left(n\frac{\log \log n}{\log n}\right) = nH₀(B) + o(n) \)
- 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
$H_0$-compressed bitvector

Space in (%) of original bitvector

bitvector = WEB-wt-1GB

- $\kappa$-array C
- $\lambda$-array O
- pointers/samples S

bitvector = DNA-wt-1GB

Space of

- $O$ blocks ending on zero
- $C$ blocks containing position $i$

Operation $\langle i \Rightarrow j \rangle$ for two, "real world" bitvectors

We try to improve on this result by applying the following optimizations:

On-the-fly decoding requires only $O(1)$ for two bitvectors. The sample rate $S$ can be answered by adding $\tilde{d} = 275$ pointers/samples $S$.

We can answer $(\tilde{i} \Rightarrow j \Rightarrow i_0)$ by sequentially scanning $C$, ... $c$

$\tilde{i}$

Note that lookup-tables for larger bitvectors are not practical. Navarro and Institute of Theoretical Informatics

The total space of $O(\log n)$ time on the fly. During the encoding process, $O(\log K)$ for two different bitvectors. The sample rate $S$ is bounded by $200$ for two, "real world" bitvectors. We can answer $\langle i \Rightarrow j \rangle$ for two, "real world" bitvectors. The sample rate $S$ can be answered by adding $\tilde{d} = 275$ pointers/samples $S$.

We can answer $(\tilde{i} \Rightarrow j \Rightarrow i_0)$ by sequentially scanning $C$, ... $c$

$\tilde{i}$

Note that lookup-tables for larger bitvectors are not practical. Navarro and Institute of Theoretical Informatics

The total space of $O(\log n)$ time on the fly. During the encoding process, $O(\log K)$ for two different bitvectors. The sample rate $S$ is bounded by $200$ for two, "real world" bitvectors. We can answer $\langle i \Rightarrow j \rangle$ for two, "real world" bitvectors. The sample rate $S$ can be answered by adding $\tilde{d} = 275$ pointers/samples $S$.

We can answer $(\tilde{i} \Rightarrow j \Rightarrow i_0)$ by sequentially scanning $C$, ... $c$

$\tilde{i}$

Note that lookup-tables for larger bitvectors are not practical. Navarro and Institute of Theoretical Informatics

The total space of $O(\log n)$ time on the fly. During the encoding process, $O(\log K)$ for two different bitvectors. The sample rate $S$ is bounded by $200$ for two, "real world" bitvectors. We can answer $\langle i \Rightarrow j \rangle$ for two, "real world" bitvectors. The sample rate $S$ can be answered by adding $\tilde{d} = 275$ pointers/samples $S$.
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 = \log_2 63).
On-the-fly block en/de-coding

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

$0 \geq 0 = \binom{0}{1}$

$0 < 1 = \binom{1}{1}$

$0 < 2 = \binom{2}{1}$

$3 \geq 3 = \binom{3}{2}$

$3 < 6 = \binom{4}{2}$

$13 \geq 10 = \binom{5}{3}$
**H₀-compressed bitvector**

![Graph](image_url)

**bitvector = WEB-WT-1GB**

- SEL-R³K
- RANK-R³K
- BV-R³K access

**bitvector = DNA-WT-1GB**

The same behavior can be observed for SOURCES-WT-200MB, ENGLISH-WT-200MB and DBLP-XML-WT-200MB. On the one hand, for bitvectors with few runs (PROTEINS-WT-200MB, DNA-WT-200MB, DNA-WT-1GB) the space usage of SEL-VS is about the same as for evenly distributed bitvectors and therefore less than SEL-C.

We independently evaluate the effect of different select implementations on the select structures. For all evaluated select structures, only one select operation is performed to determine the position of the i-th bit in a target word x. Before this step, potentially many other operations such as popcnts in a sequential scan or binary search are performed to determine x. We measure the effect of using a slow select 64 method (TBL) compared the fastest method (BLT) of the target word on the overall performance of the structure. For small instances, the running time for SEL-C and SEL-VS decreases by 10 to 20%, for the binary search structures it decreased by 7%. For large instances, the running time for SEL-C and SEL-VS decreases only by 2% to 5%. The binary search solutions do not benefit from improved select 64 on large instances as the running time is dominated by binary search and the resulting TLB misses. Interestingly, the running time for SEL-V9 decreases by 30% for all instances while all other select structures improve mostly for small instances.

Another consideration when choosing a select structure is construction cost. We use both popcnt and select 64 methods during the construction of our SEL-C structure. We observed an improvement in construction cost by up to an order of magnitude (up to 30 times faster for the random bitvectors used in our experiments) compared to versions with simple bit-by-bit processing during construction.
$\mathcal{H}_0$-compression for sequences

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

$$\mathcal{H}_0(S) = \sum_{c \in [0, \sigma - 1]} \frac{n_c}{n} \log \frac{n}{n_c}$$

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

Idea

Represent $S$ as a wavelet tree and use $\mathcal{H}_0$-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 $v_0$ represents the whole sequence $S_{v_0} = S$. 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 $S_{i(2)}$. 
$\mathcal{H}_0$-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 $n_{\omega 0} \mathcal{H}_0(S_{\omega 0})$ and $n_{\omega 1} \mathcal{H}_0(S_{\omega 0})$. The space to represent $S_\omega$ is

$$= n_{\omega} \mathcal{H}_0(B_\omega) + n_{\omega 0} \mathcal{H}_0(S_{\omega 0}) + n_{\omega 1} \mathcal{H}_0(S_{\omega 1})$$

$$= n_{\omega 0} \cdot \log \frac{n_\omega}{n_{\omega 0}} + n_{\omega 1} \cdot \log \frac{n_\omega}{n_{\omega 1}} + n_{\omega 0} \mathcal{H}_0(S_{\omega 0}) + n_{\omega 1} \mathcal{H}_0(S_{\omega 1})$$

$$= n_{\omega 0} \cdot \log \frac{n_\omega}{n_{\omega 0}} + n_{\omega 0} \mathcal{H}_0(S_{\omega 0}) + n_{\omega 1} \cdot \log \frac{n_\omega}{n_{\omega 1}} + n_{\omega 1} \mathcal{H}_0(S_{\omega 1})$$

(a) 

(b)

For (a) we get with the definition of $n_{\omega 0} \mathcal{H}_0(S_{\omega 0}) = \sum_{\alpha \in \sigma^{H-L}} n_{\omega 0 \alpha} \log \frac{n_{\omega 0 \alpha}}{n_{\omega 0}}$

(a)  

$$= \sum_{\alpha \in \sigma^{H-L}} n_{\omega 0 \alpha} \left( \log \frac{n_\omega}{n_{\omega 0}} + \log \frac{n_{\omega 0}}{n_{\omega 0 \alpha}} \right) = \sum_{\alpha \in \sigma^{H-L}} n_{\omega 0 \alpha} \log \frac{n_\omega}{n_{\omega 0 \alpha}}$$
**H₀**-compression for sequences

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

\[
\begin{align*}
    &= \sum_{\alpha \in \sigma^{H-L}} n_{\omega_0\alpha} \log \frac{n_\omega}{n_{\omega_0\alpha}} + \sum_{\alpha \in \sigma^{H-L}} n_{\omega_1\alpha} \log \frac{n_\omega}{n_{\omega_1\alpha}} \\
    &= \sum_{\alpha' \in \sigma^{H-(L-1)}} n_{\omega_\alpha'} \log \frac{n_\omega}{n_{\omega_\alpha'}} \\
    &= n_\omega \mathcal{H}_0(S_\omega)
\end{align*}
\]

Induction start for \( L = H \) (leaf nodes of WT). For a single symbol \( \omega' \in \Sigma \) we get \( \mathcal{H}_0(S_{\omega'}) = 0 \).
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
$\mathcal{H}_k$ 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>
Question: How can we adjust the FM-index to use just $\mathcal{H}_k(T) + O(\sigma^k)$ bits of space?


