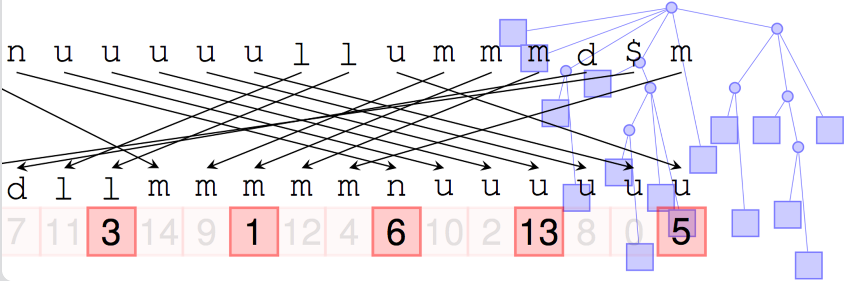
Karlsruhe Institute of Technology

Text Indexing: Lecture 2

Simon Gog – *gog@kit.edu*

Institute of Theoretical Informatics - Algorithmics



Navigating in the text via Ψ and LF

i	LF	T^{BWT}	F
0	4	m	$
1	9	n	dumulmum$
2	10	u	lmum$
3	11	u	lmundumulmum$
4	12	u	m$
5	13	u	mulmum$
6	14	u	mulmundumulmum$
7	2	l	mum$
8	3	l	mundumulmum$
9	15	u	ndumulmum$
10	5	m	ulmum$
11	6	m	ulmundumulmum$
12	7	m	um$
13	1	d	umulmum$
14	0	$	umulmundumulmum$
15	8	m	undumulmum$

(a)

Ψ	T^{BWT}	F
14	m	$
13	n	dumulmum$
7	u	lmum$
8	u	lmundumulmum$
0	u	m$
10	u	mulmum$
11	u	mulmundumulmum$
12	l	mum$
15	l	mundumulmum$
1	u	ndumulmum$
2	m	ulmum$
3	m	ulmundumulmum$
4	m	um$
5	d	umulmum$
6	$	umulmundumulmum$
9	m	undumulmum$

(b)

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

i	*SA*	*ISA*	
0	15	14	$
1	7	6	dumulmum$
2	11	11	lmum$
3	3	3	lmundumulmum$
4	14	8	m$
5	9	15	mulmum$
6	1	9	mulmundumulmum$
7	12	1	mum$
8	4	13	mundumulmum$
9	6	5	ndumulmum$
10	10	10	ulmum$
11	2	2	ulmundumulmum$
12	13	7	um$
13	8	12	umulmum$
14	0	4	umulmundumulmum$
15	5	0	undumulmum$

- 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 Ψ:

$$\Psi[i] = ISA[SA[i] + 1 \mod n]$$

Institute of Theoretical Informatics
Algorithmics

Implement Ψ via WT over BWT

Ψ calculation

$\Psi[i] = select(rank(i, F[i], F), F[i], BWT)$

Operation select

Given a sequence X, a symbol c, and an integer i. Operation $select(i, c, X)$ 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 σ.
- What is the maximal size of the set $\{i \mid \Psi[i] > \Psi[i+1]\}$?

Turning the FM-Index into a Self-Index

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 $B[i] = 1$, $SA'[rank(i, 1, B)] = SA[i]$.

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 Σ. 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.

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 $(= \#zeros)$ 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.

Compressing the Index

How does Elias-Fano coding work?

$$X = \quad 4 \quad 13 \quad 15 \quad 24 \quad 26 \quad 27 \quad 29$$

$$\underbrace{00}_{0}\underbrace{100}_{4} \quad \underbrace{01}_{1}\underbrace{101}_{5} \quad \underbrace{01}_{1}\underbrace{111}_{7} \quad \underbrace{11}_{3}\underbrace{000}_{0} \quad \underbrace{11}_{3}\underbrace{010}_{2} \quad \underbrace{11}_{3}\underbrace{011}_{3} \quad \underbrace{11}_{3}\underbrace{101}_{5}$$

0-0 1-0 1-1 3-1 3-3 3-3 3-3

$$\delta = \quad 0 \quad 1 \quad 0 \quad 2 \quad 0 \quad 0 \quad 0$$

$$H = 1011001111$$

$$L = \quad 4 \quad 5 \quad 7 \quad 0 \quad 2 \quad 3 \quad 5$$

Compressing the Index

How does Elias-Fano coding work?

Constant time access

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

$access(i)$
  $p \leftarrow select(i+1, 1, H)$
  $x \leftarrow p - i$
  return $x \cdot 2^{\lceil \log n \rceil - \lfloor \log m \rfloor} + L[i]$

Compressing the Index

Apply Elias-Fano coding to a Ψ-based CSA

- Ψ consists of at most σ non-decreasing sequences in the range $[0, n-1]$.
-

$$\begin{aligned} |CSA_{\Psi}| &= \sum_{c\in\Sigma}\left(n_c(2+\log\frac{n}{n_c}) + o(n_c)\right) \\ &= \sum_{c\in\Sigma} 2n_c + n\sum_{c\in\Sigma}\frac{n_c}{n}\log\frac{n}{n_c} + o(n) \\ &= 2n + nH_0(\mathcal{T}) + o(n) \end{aligned}$$

- $+O(\sigma \log n)$ bits to handle character boundaries

Compressing the Index

Search in a Ψ-based CSA

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

compare(P, i)
    $k \leftarrow 0$
    while $k < |P|$ **do**
        if $C[P[k]+1]-1 < i$ **then**
            return -1 //P smaller than suffix
        else if $C[P[k]] > i$ **then**
            return $+1$ //P larger than suffix
        $k \leftarrow k+1$
        $i \leftarrow \Psi[i]$
    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 $|bin(|bin(x)|)| - 1$ in unary, append the $|bin(|bin(x)|)| - 1$ least significant bits of $|bin(x)|$, and append the $|bin(x)| - 1$ least significant bits of $bin(x)$.

$x_{(10)}$	$x_{(unary)}$	$x_{(2)}$	$x_{(\delta-code)}$	$\lvert x_{\delta-code}\rvert$
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	00000000000001	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 Ψ-based CSA using Elias-δ code.

For each character c gap-encode its increasing Ψ 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 $i = 0$.

$$\begin{aligned}
& \sum_{c\in\Sigma} \sum_{i=0}^{n_c-1} (\log g_{c,i} + 2\log\log g_{c,i} + O(1)) \\
\leq\ & O(n) + \sum_{c\in\Sigma} \sum_{i=0}^{n_c-1} \left(\log \frac{n}{n_c} + 2\log\log \frac{n}{n_c}\right) \\
=\ & O(n) + n \sum_{c\in\Sigma} \frac{n_c}{n} \left(\log \frac{n}{n_c} + 2\log\log \frac{n}{n_c}\right) \\
=\ & nH_0(T) + O(n\log\log n)
\end{aligned}$$

Compressed Bitvectors

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 represenations:

- Elias-Fano coded bitvector (`sd_vector`);
  see Okanohara & Sadakane [4]
- H_0-compressed bitvector (`rrr_vector`); see Raman et al. [5]

Elias-Fano coded bitvector

Let B be a bitvector of length n and κ 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 κ= # 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 $\mathcal{O}(n\frac{\log\log n}{\log n})$ bits

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

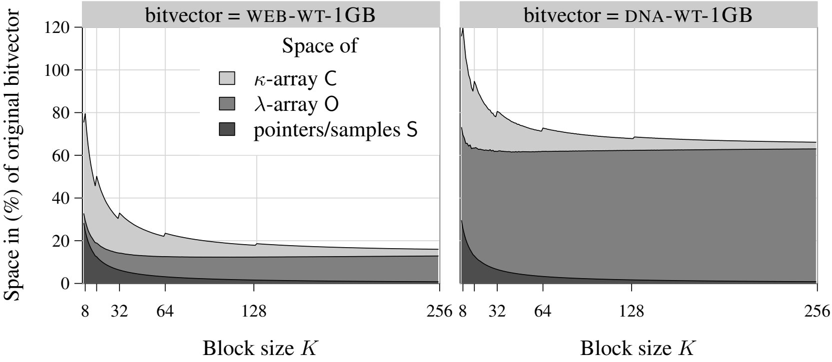
$$\left\lceil \log \binom{K}{\kappa_0} \right\rceil + \cdots + \left\lceil \log \binom{K}{\kappa_{(n-1)/K}} \right\rceil \leq \log \left(\binom{K}{\kappa_0} \times \cdots \times \binom{K}{\kappa_{(n-1)/K}} \right) + n/K$$

$$\leq \log \binom{n}{\kappa_0 + \cdots + \kappa_{(n-1)/K}} + n/K = \log \binom{n}{\kappa} + n/K = nH_0(B) + \mathcal{O}(n/\log n)$$

H_0-compressed bitvector

- Lookup table to map between class indexes and block
- Overall space: $nH_0(B) + \mathcal{O}(\frac{n}{\log n}) + \mathcal{O}(n\frac{\log\log n}{\log n}) = nH_0(B) + o(n)$
- 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

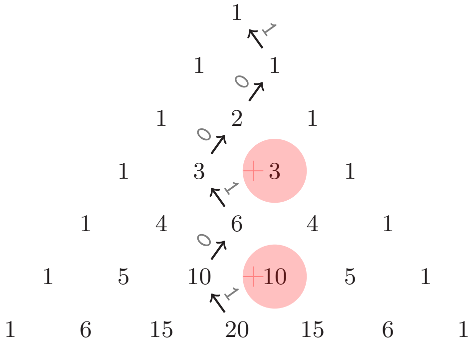
H_0-compressed bitvector



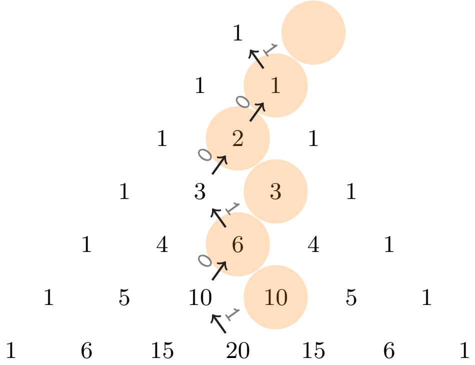
On-the-fly block en/de-coding

- Use combinatorial number system of degree κ_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



On-the-fly block en/de-coding



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

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

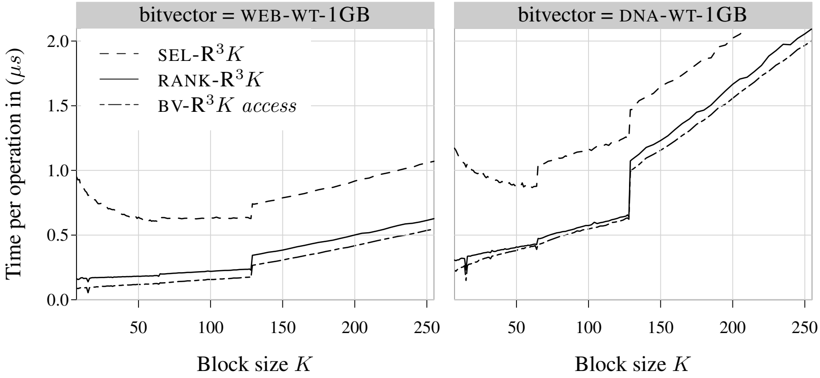
$0 < 2 = \binom{2}{1}$

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

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

$13 \geq 10 = \binom{5}{3}$

H_0-compressed bitvector



H_0-compression for sequences

Let S be a sequence of length n over alphabet Σ of size σ.

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

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

H_0-compression for sequences

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_ϵ 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 $v_{t_{(2)}}$ and $B_{i_{(2)}}$ the bitvector which consists of the ℓ-th bits of S_{i_2}.

H_0-compression for sequences

Let ω 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 the space to represent S_ω is

$$
\begin{aligned}
&= n_\omega H_0(B_\omega) + n_{\omega 0} H_0(S_{\omega 0}) + n_{\omega 1} H_0(S_{\omega 1}) \\
&= n_{\omega 0} \cdot \log \frac{n_\omega}{n_{\omega 0}} + n_{\omega 1} \cdot \log \frac{n_\omega}{n_{\omega 1}} + n_{\omega 0} H_0(S_{\omega 0}) + n_{\omega 1} H_0(S_{\omega 1}) \\
&= \underbrace{n_{\omega 0} \cdot \log \frac{n_\omega}{n_{\omega 0}} + n_{\omega 0} H_0(S_{\omega 0})}_{(a)} + \underbrace{n_{\omega 1} \cdot \log \frac{n_\omega}{n_{\omega 1}} + n_{\omega 1} H_0(S_{\omega 1})}_{(b)}
\end{aligned}
$$

For (a) we get with the definition of $n_{\omega 0}H_0(S_{\omega 0}) = \sum_{\alpha \in \sigma^{H-L}} n_{\omega 0\alpha} \log \frac{n_{\omega 0}}{n_{\omega 0\alpha}}$

$$
(a) \quad = \sum_{\alpha \in \sigma^{H-L}} n_{\omega 0\alpha} \left(\log \frac{n_\omega}{n_{\omega 0}} + \log \frac{n_{\omega 0}}{n_{\omega 0\alpha}} \right) = \sum_{\alpha \in \sigma^{H-L}} n_{\omega 0\alpha} \log \frac{n_\omega}{n_{\omega 0\alpha}}
$$

H_0-compression for sequences

Space to represnt S_ω by adding (a) and (b)

$$
\begin{aligned}
&= \sum_{\alpha \in \sigma^{H-L}} n_{\omega 0 \alpha} \log \frac{n_\omega}{n_{\omega 0 \alpha}} + \sum_{\alpha \in \sigma^{H-L}} n_{\omega 1 \alpha} \log \frac{n_\omega}{n_{\omega 1 \alpha}} \\
&= \sum_{\alpha' \in \sigma^{H-L-1}} n_{\omega \alpha'} \log \frac{n_\omega}{n_{\omega \alpha'}} \\
&= n_\omega H_0(S_\omega)
\end{aligned}
$$

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 kth order entropy is defined as

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

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

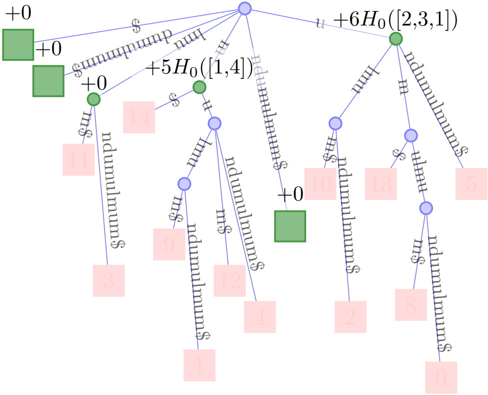
$C = \{\texttt{an}, \texttt{na}, \texttt{as}\}$
$S_{\texttt{an}} = \texttt{aa}, S_{\texttt{na}} = \texttt{ns}, S_{\texttt{as}} = \epsilon$
$\rightarrow H_2(T) = \frac{2}{6} H_0(\texttt{ns}) = \frac{1}{3}$ bits

H_k of *Pizza&Chili* corpus texts (200MB versions)

$H_k(T)$ in bits contexts/$|T|$ in percent

k	DBLP.XML		DNA		ENGLISH		PROTEINS	
0	5.26	0.0	1.97	0.0	4.53	0.0	4.20	0.0
1	3.48	0.0	1.93	0.0	3.62	0.0	4.18	0.0
2	2.17	0.0	1.92	0.0	2.95	0.0	4.16	0.0
3	1.43	0.1	1.92	0.0	2.42	0.0	4.07	0.0
4	1.05	0.4	1.91	0.0	2.06	0.3	3.83	0.1
5	0.82	1.3	1.90	0.0	1.84	1.0	3.16	1.7
6	0.70	2.7	1.88	0.0	1.67	2.7	1.50	17.4

H_k compression



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

Bibliography

[1] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, April 1974.

[2] Robert Mario Fano. *On the Number of Bits Required to Implement an Associative Memory*. Computation Structures Group Memo. MIT, Project MAC, 1971.

[3] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Proc. SEA*, pages 295–306, 2012.

[4] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX*, 2007.

[5] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA*, pages 233–242, 2002.