

Text Indexing

Lecture 07: FM-Index and r -Index

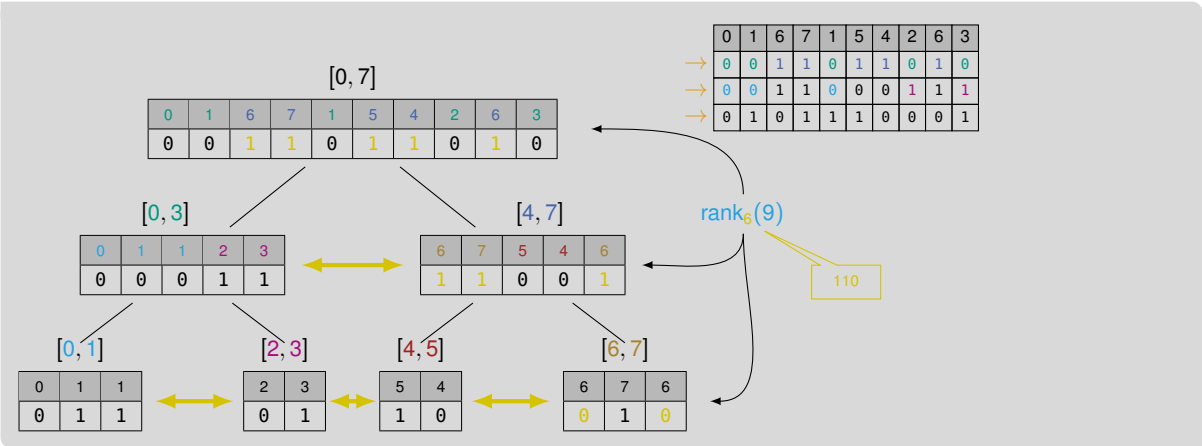
Florian Kurpicz

The slides are licensed under a Creative Commons Attribution-ShareAlike 4.0 International License © ⓘ ⓘ: www.creativecommons.org/licenses/by-sa/4.0 | commit 224e27c compiled at 2022-12-12-13:24



<https://pingo.scc.kit.edu/022183>

Recap: Wavelet Trees




Recap: Compressed Wavelet Trees

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

- intervals are only missing to the right (white space)
- no holes allow for easy querying

- build wavelet tree for compressed text
- compress text using bit-wise negated canonical Huffman-codes

- can a wavelet tree be compressed further?
-  **PINGO** are there compressed bit vectors with $O(1)$ access time?

Bit Vector Compression (1/2)

- compress (sparse) bit vectors
- bit vector contains k one bits
- use $O(k \lg \frac{n}{k}) + o(n)$ bits
- retrieve $\Theta(\lg n)$ bits at the same time
- similar to *rank* data structure

- split bit vector into (super-)blocks
- blocks of size $s = \frac{\lg n}{2}$
- super-blocks of size $s' = s^2$

Array C

- number of ones in i -th block

Lookup-Tables L_i

- for $i \in [0, s]$ store lookup-table containing all bit vectors with i one bits

- use variable-length codes to identify content of block
- concatenate all codes in bit vector V

Bit Vector V

- let k_i be number of ones in i -th block
- use $\lceil \lg \binom{s}{k_i} \rceil$ bits to encode block i position in lookup-table
- concatenate all codes

Bit Vector Compression (2/2)

Array *SBlock*

- for every super-block i , $SBlock[i]$ contains position of encoding of first block in i -th super-block in V
- $\lceil \lg n \rceil$ bits per entry

Array *Block*

- for every block i , $Block[i]$ contains position of encoding of i -th block in V relative to its super-block
- $O(\lg \lg n)$ bits per entry

Lemma: Compressed Bit Vectors

A bit vector of size n containing k ones can be represented using $O(k \lg \frac{n}{k}) + o(n)$ bits allowing $O(1)$ time access to individual bits

Proof (Sketch space requirements)


- $|C| = O(\frac{n}{s} \lg s) = o(n)$ bits
- $|SBlock| = O(\frac{n}{s'} \lg n) = o(n)$ bits
- $|Block| = O(\frac{n}{s} \lg s) = o(n)$ bits
- $\sum_{k=0}^s |L_k| \leq (s+1)2^s s = o(n)$ bits
- $|V| = \sum_{i=1}^{\lceil n/s \rceil} \lceil \lg \binom{s}{k_i} \rceil \leq \lg \binom{n}{k} + n/s \leq \lg((n/k)^k) + n/s = k \lg \frac{n}{k} + O(\frac{n}{\lg n})$ bits

Recap: Backwards Search in the BWT

Function *BackwardsSearch*($P[1..n]$, C , $rank$):

```

1  |  $s = 1, e = n$ 
2  | for  $i = m, \dots, 1$  do
3  |   |  $s = C[P[i]] + rank_{P[i]}(s - 1) + 1$ 
4  |   |  $e = C[P[i]] + rank_{P[i]}(e)$ 
5  |   | if  $s > e$  then
6  |   |   | return  $\emptyset$ 
7  | return  $[s, e]$ 
  
```

- 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 

The FM-Index [FM00]

Building Blocks of FM-Index

- wavelet tree on BWT providing *rank*-function
- *C*-array
- sampled suffix array with sample rate s
- bit vector marking sampled suffix array positions

Lemma: FM-Index

Given a text T of length n over an alphabet of size σ , the FM-index requires $O(n \lg \sigma)$ bits of space and can answer counting queries in $O(m \lg \sigma)$ time and reporting queries in $O(occ + m \lg \sigma)$ time

Space Requirements

- wavelet tree: $n \lceil \lg \sigma \rceil (1 + o(1))$ bits
 - *C*-array: $\sigma \lceil \lg n \rceil$ bits $\ominus n(1 + o(1))$ 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
- space and time bounds can be achieved with $s = \lg_{\sigma} n$


Conclusion FM-Index

- FM-index is easy to compress
 - wavelet tree on *BWT* can be compressed
 - bit vector can be compressed
-
- very small in comparison with suffix tree or suffix array
 - compression does not make use of structure of *BWT* ⓘ wavelet trees are compressed using Huffman-codes

Definition: Run (simplified, recap)

Given a text T of length n , we call its substring $T[i..j]$ a **run**, if

- $T[k] = T[l]$ for all $k, l \in [i, j]$ 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 )

	1	2	3	4	5	6	7	8	9	0	11	12	13
L	a	b	\$	c	c	b	b	a	a	a	a	b	b

Motivation: r -Index

- next: compressed index
- how to measure compressibility?

Measure for Compressibility

- k -th order empirical entropy H_k
 - number of LZ factors z
 - number of *BWT* runs r
-
- z and r not blind to repetitions
 - how do they relate?

Lemma: *BWT* runs and LZ factors [KK20]

Given a text T of length n . Let z be the number of LZ77 factors and r the number of runs in T 's *BWT*, then

$$r \in O(z \lg^2 n)$$

- more details in next lecture

Main Part of Backwards-Search


Function *BackwardsSearch*($P[1..n]$, C , $rank$):

```
1 |  $s = 1, e = n$ 
2 | for  $i = m, \dots, 1$  do
3 | |  $s = C[P[i]] + rank_{P[i]}(s - 1) + 1$ 
4 | |  $e = C[P[i]] + rank_{P[i]}(e)$ 
5 | | if  $s > e$  then
6 | | | return  $\emptyset$ 
7 | return  $[s, e]$ 
```

Goals

- simulate *BWT* and *rank* on *BWT* in
- $O(r \lg n)$ bits of space

The r -Index [GNP20] (1/3)

Given a text T of length n over an alphabet Σ and its BWT , the r -index of this text consists of the following data structures 

Array I

- $I[i]$ stores position of i -th run in BWT

Array L'

- $L'[i]$ stores character of i -th run in BWT
- build wavelet tree for L'

Array R

- lengths of BWT runs stably sorted by runs' characters
- accumulate for each character by performing exclusive prefix sum over run lengths'

Array C'

- $C'[\alpha]$ stores the start of the run lengths in R for each character $\alpha \in \Sigma$ starting at 0

Bit Vector B

- compressed bit vector of length n containing ones at positions where BWT runs start and rank-support

The r -Index (2/3)

$rank_{\alpha}(BWT, i)$ with r -Index

- compute number j of run ($j = rank_1(B, i)$)
- compute position k in R ($k = C'[\alpha]$)
- compute number ℓ of α runs before the j -th run ($\ell = rank_{\alpha}(L', j - 1)$)
- compute number k of α s before the j -th run ($k = R[k + \ell]$)
- compute character β of run ($\beta = L'[j]$)
- if $\alpha \neq \beta$ return k ⓘ i is not in the run
- else return $k + i - l[j] + 1$ ⓘ i is in the run

The r -Index (3/3)

Lemma: Space Requirements r -Index

Given a text T of length n over an alphabet of size σ that has r BWT runs, then its r -index requires

$$O(r \lg n) \text{ bits}$$

and can answer *rank*-queries on the BWT in $O(\lg \sigma)$.
Given a pattern of length m , the r -index can answer pattern matching queries in time


$$O(m \lg \sigma)$$

- what about reporting queries?


Locating Occurrences (Sketch)


- modify backwards-search that it maintains $SA[e]$
- after backwards-search output $SA[e], SA[e - 1], \dots, SA[s]$
- in $O(r \lg n)$ bits and $O(occ \cdot \lg \lg r)$ time

Maintaining $SA[e]$

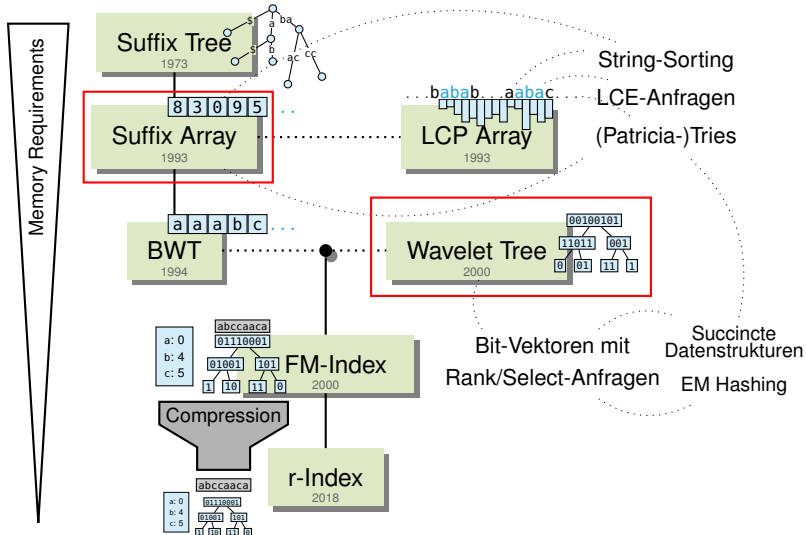
- sample SA positions at ends of runs
- if next character is $BWT[e]$, then next $SA[e']$ is $SA[e] - 1$
- otherwise locate end of run and extract sample 

Output Result

- following LF not possible  unbounded
- deduce $SA[i - 1]$ from $SA[i]$
- character in L and F in same order
- only beginning of runs complicated
- for every character build predecessor data structure over sampled SA -values at end of runs
- associate with $\langle i, SA[i] \rangle$

-  **PINGO** why can't we sample the SA as we did in the FM-index?

From the Suffix Tree to the *r*-Index—Questions?



Bibliography I

- [FM00] Paolo Ferragina and Giovanni Manzini. “Opportunistic Data Structures with Applications”. In: *FOCS*. IEEE Computer Society, 2000, pages 390–398. DOI: [10.1109/SFCS.2000.892127](https://doi.org/10.1109/SFCS.2000.892127).
- [GNP20] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. “Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space”. In: *J. ACM* 67.1 (2020), 2:1–2:54. DOI: [10.1145/3375890](https://doi.org/10.1145/3375890).
- [KK20] Dominik Kempa and Tomasz Kociumaka. “Resolution of the Burrows-Wheeler Transform Conjecture”. In: *FOCS*. IEEE, 2020, pages 1002–1013. DOI: [10.1109/FOCS46700.2020.00097](https://doi.org/10.1109/FOCS46700.2020.00097).