MOVE-R: OPTIMIZING THE R-INDEX

Symposium on Experimental Algorithms 2024 ·
Nico Bertram, Johannes Fischer and Lukas Nalbach

Text Indexing

Text Indexing

- Text index: data structure for a string T to answer **count** and **locate** queries

Text Indexing

Text Indexing

- Text index: data structure for a string T to answer **count** and **locate** queries
- Count query: How often does the pattern P occur in T?
- Locate query: At which positions does the pattern P occur in T?

Text Indexing

Text Indexing

- Text index: data structure for a string T to answer **count** and **locate** queries
- Count query: How often does the pattern P occur in T?
- Locate query: At which positions does the pattern P occur in T?

Queries

- $T = acbbcacbc$

Text Indexing

Text Indexing

- Text index: data structure for a string T to answer **count** and **locate** queries
- Count query: How often does the pattern P occur in T?
- Locate query: At which positions does the pattern P occur in T?

Queries

- $T = acb\underline{bc}ac\underline{bc}$
- **count**$(\underline{bc}) = 2$

Text Indexing

Text Indexing

- Text index: data structure for a string T to answer **count** and **locate** queries
- Count query: How often does the pattern P occur in T?
- Locate query: At which positions does the pattern P occur in T?

Queries

- $T = acbbcacbc$
- **count**$(bc) = 2$
- **locate**$(ac) = \{1, 6\}$

Text Indexing

Text Indexing

- Text index: data structure for a string T to answer **count** and **locate** queries
- Count query: How often does the pattern P occur in T?
- Locate query: At which positions does the pattern P occur in T?
- **Compressed** Text Index: utilizes information redundancy in **repetitive** strings $\Longrightarrow$ **lower memory footprint**

Repetitive Strings

- $T_1 = bbccaaaaccbbaaaa$
- $T_2 = \mathtt{ATCGATCGATCGAT}$

Queries

- $T = acbbcacbc$
- **count**$(bc) = 2$
- **locate**$(ac) = \{1, 6\}$

Text Indexing

Text Indexing

- Text index: data structure for a string T to answer **count** and **locate** queries
- Count query: How often does the pattern P occur in T?
- Locate query: At which positions does the pattern P occur in T?
- **Compressed** Text Index: utilizes information redundancy in **repetitive** strings $\Longrightarrow$ **lower memory footprint**

Repetitive Strings

- $T_1 = bbccaaaaccbbaaaa$
- $T_2 = \mathtt{ATCGATCGATCGAT}$
- in practice: DNA, log files, versioned documents, natural language

Queries

- $T = acbbcacbc$
- **count**$(bc) = 2$
- **locate**$(ac) = \{1, 6\}$

Text Indexing

BWT-based Text Indexes

- Text T of length n
- Suffix $T_i = T[i, n]$

Text Indexing

BWT-based Text Indexes

- Text T of length n
- Suffix $T_i = T[i, n]$
- Lexicographical order $\prec$ (i.e. $abc \prec acb$)
- Suffix array $\mathrm{SA}[1..n]$, s.t. $T_{\mathrm{SA}[1]} \prec \ldots \prec T_{\mathrm{SA}[n]}$

Burrows Wheeler Matrix (BWM)

- $T = acbcbac\$$

i	SA	
1	8	
2	6	
3	1	
4	5	
5	3	
6	7	
7	4	
8	2	

Text Indexing

BWT-based Text Indexes

- Text T of length n
- Suffix $T_i = T[i, n]$
- Lexicographical order $\prec$ (i.e. $abc \prec acb$)
- Suffix array $\mathrm{SA}[1..n]$, s.t. $T_{\mathrm{SA}[1]} \prec \ldots \prec T_{\mathrm{SA}[n]}$
- $\mathrm{rot}(T, i) = T[i, n]T[1, i)$

Burrows Wheeler Matrix (BWM)

- $T = acbcbac\$$
- i-th row = $\mathrm{rot}(T, \mathrm{SA}[i])$

i	SA	F		L
1	8	$	acbcba	c
2	6	a	c$acbc	b
3	1	a	cbcbac	$
4	5	b	ac$acb	c
5	3	b	cbac$a	c
6	7	c	$acbcb	a
7	4	c	bac$ac	b
8	2	c	bcbac$	a

Text Indexing

BWT-based Text Indexes

- Text T of length n
- Suffix $T_i = T[i, n]$
- Lexicographical order $\prec$ (i.e. $abc \prec acb$)
- Suffix array $\mathsf{SA}[1..n]$, s.t. $T_{\mathsf{SA}[1]} \prec \ldots \prec T_{\mathsf{SA}[n]}$
- $\mathrm{rot}(T, i) = T[i, n]T[1, i)$
- Burrows Wheeler Transform (BWT) = last (**L**) column of the BWM

Burrows Wheeler Matrix (BWM)

- $T = acbcbac\$$
- i-th row = $\mathrm{rot}(T, \mathsf{SA}[i])$

i	SA	F		L
1	8	$	acbcba	c
2	6	a	c$acbc	b
3	1	a	cbcbac	$
4	5	b	ac$acb	c
5	3	b	cbac$a	c
6	7	c	$acbcb	a
7	4	c	bac$ac	b
8	2	c	bcbac$	a

Text Indexing

BWT-based Text Indexes

- Text T of length n
- Suffix $T_i = T[i, n]$
- Lexicographical order $\prec$ (i.e. $abc \prec acb$)
- Suffix array $\mathrm{SA}[1..n]$, s.t. $T_{\mathrm{SA}[1]} \prec \ldots \prec T_{\mathrm{SA}[n]}$
- $\mathrm{rot}(T, i) = T[i, n]T[1, i)$
- Burrows Wheeler Transform (BWT) = last (L) column of the BWM
- SA-interval $[b, e]$ of P stores occurrences of P in T

SA-Interval

- $P = ac$ has SA-interval $[2, 3]$

$\Longrightarrow$ P occurs at $\mathrm{SA}[2, 3] = [6, 1]$ in T

Burrows Wheeler Matrix (BWM)

- $T = acbcbac\$$
- i-th row = $\mathrm{rot}(T, \mathrm{SA}[i])$

i	SA	F		L
1	8	$	acbcba	c
2	$\boxed{6}$	a	c$acbc	b
3	$\boxed{1}$	a	cbcbac	$
4	5	b	ac$acb	c
5	3	b	cbac$a	c
6	7	c	$acbcb	a
7	4	c	bac$ac	b
8	2	c	bcbac$	a

Text Indexing

Compressed BWT-based Text Indexes

- Let r = # equal-letter runs in L, σ = # distinct characters in T, ω = word-width of the word-RAM, m = length of the pattern
- r-index [6]:
  - $\mathcal{O}(r)$ space
  - Implements functions LF and Φ in $\underline{\mathcal{O}(\log\log_\omega n/r)\text{ time}}$
  - Count: $\mathcal{O}(m \log\log_\omega(\sigma + n/r))$ time
  - Locate: additional $\mathcal{O}(occ \log\log_\omega(n/r))$ time

Burrows Wheeler Matrix (BWM)

- $T = acbcbac\$$
- i-th row = $\text{rot}(T, \text{SA}[i])$

i	SA	F		L
1	8	$	acbcba	c
2	6	a	c$acbc	b
3	1	a	cbcbac	$
4	5	b	ac$acb	c
5	3	b	cbac$a	c
6	7	c	$acbcb	a
7	4	c	bac$ac	b
8	2	c	bcbac$	a

Text Indexing

Compressed BWT-based Text Indexes

- Let r = # equal-letter runs in L, σ = # distinct characters in T, ω = word-width of the word-RAM, m = length of the pattern
- r-index [6]:
  - $\mathcal{O}(r)$ space
  - Implements functions LF and Φ in $\underline{\mathcal{O}(\log \log_{\omega} n/r)}$ time
  - Count: $\mathcal{O}(m \log \log_{\omega}(\sigma + n/r))$ time
  - Locate: additional $\mathcal{O}(occ \log \log_{\omega}(n/r))$ time
- OptBWTR [8] (not yet implemented):
  - $\mathcal{O}(r)$ space
  - Implements functions LF and Φ in $\underline{\mathcal{O}(1)}$ time using <u>move data structures</u>
  - Count: $\mathcal{O}(m \log \log_{\omega} \sigma)$ time
  - Locate: additional $\mathcal{O}(occ)$ time

Burrows Wheeler Matrix (BWM)

- $T = acbcbac\$$
- i-th row = $\mathrm{rot}(T, \mathrm{SA}[i])$

i	SA	F		L
1	8	\$	acbcba	c
2	6	a	c\$acbc	b
3	1	a	cbcbac	\$
4	5	b	ac\$acb	c
5	3	b	cbac\$a	c
6	7	c	\$acbcb	a
7	4	c	bac\$ac	b
8	2	c	bcbac\$	a

Text Indexing

Compressed BWT-based Text Indexes

- Let r = # equal-letter runs in L, σ = # distinct characters in T, ω = word-width of the word-RAM, m = length of the pattern
- r-index [6]:
  - $\mathcal{O}(r)$ space
  - Implements functions LF and Φ in $\mathcal{O}(\log\log_\omega n/r)$ time
  - Count: $\mathcal{O}(m \log\log_\omega(\sigma + n/r))$ time
  - Locate: additional $\mathcal{O}(occ \log\log_\omega(n/r))$ time
- OptBWTR [8] (not yet implemented):
  - $\mathcal{O}(r)$ space
  - Implements functions LF and Φ in $\mathcal{O}(1)$ time using move data structures
  - Count: $\mathcal{O}(m \log\log_\omega \sigma)$ time
  - Locate: additional $\mathcal{O}(occ)$ time
- Is the improved time complexity reflected in practice?

Burrows Wheeler Matrix (BWM)

- $T = acbcbac\$$
- i-th row = $\mathrm{rot}(T, \mathrm{SA}[i])$

i	SA	F		L
1	8	$	acbcba	c
2	6	a	c$acbc	b
3	1	a	cbcbac	$
4	5	b	ac$acb	c
5	3	b	cbac$a	c
6	7	c	$acbcb	a
7	4	c	bac$ac	b
8	2	c	bcbac$	a

Text Indexing

Our Contribution

- Move-r: practically optimized implementation of OptBWTR
  - Practically optimized implementation and construction of the move data structure and other index data structures
  - Practically optimized count- and locate algorithms
  - More optimizations

Burrows Wheeler Matrix (BWM)

- $T = acbcbac\$$
- i-th row = $\mathrm{rot}(T, \mathrm{SA}[i])$

i	SA	F		L
1	8	$	acbcba	c
2	6	a	c$acbc	b
3	1	a	cbcbac	$
4	5	b	ac$acb	c
5	3	b	cbac$a	c
6	7	c	$acbcb	a
7	4	c	bac$ac	b
8	2	c	bcbac$	a

Text Indexing

Our Contribution

- Move-r: practically optimized implementation of OptBWTR
  - Practically optimized implementation and construction of the move data structure and other index data structures
  - Practically optimized count- and locate algorithms
  - More optimizations
- Compared with the resp. fastest other index:
  - 2x-35x (typ. 15x) faster queries
  - 0.8x-2.5x (typ. 2x) larger index
  - 0.9-2x (typ. 2x) faster construction with 1-3 (typ. 3x) lower memory usage

Burrows Wheeler Matrix (BWM)

- $T = acbcbac\$$
- i-th row = $\mathrm{rot}(T, \mathrm{SA}[i])$

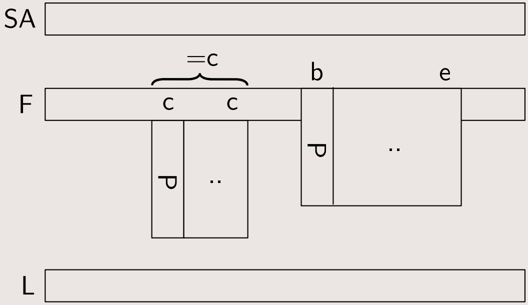
i	SA	F		L
1	8	\$	acbcba	c
2	6	a	c\$acbc	b
3	1	a	cbcbac	\$
4	5	b	ac\$acb	c
5	3	b	cbac\$a	c
6	7	c	\$acbcb	a
7	4	c	bac\$ac	b
8	2	c	bcbac\$	a

Compressed Text Indexes

Backward Search (Step)

- Given SA-interval $[b, e]$ of P
- Compute SA-interval of cP

Illustration

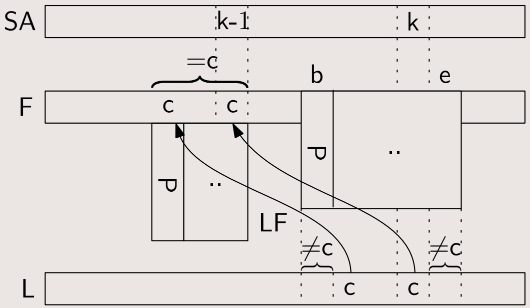


Compressed Text Indexes

Backward Search (Step)

- Given SA-interval $[b, e]$ of P
- Compute SA-interval of cP
- $\mathrm{LF}(i)$ = position of $\mathrm{SA}[i] - 1$ in SA (shift rotation down by 1)

Illustration



Compressed Text Indexes

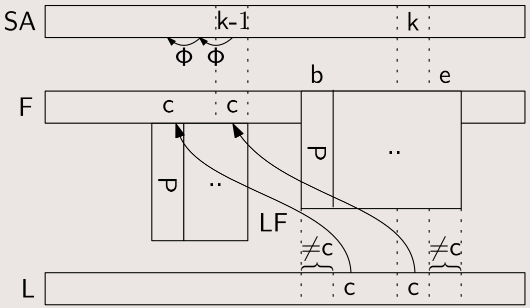
Backward Search (Step)

- Given SA-interval $[b, e]$ of P
- Compute SA-interval of cP
- $\mathrm{LF}(i)$ = position of $\mathrm{SA}[i] - 1$ in SA (shift rotation down by 1)

Locate Query

- Compute values of SA in the SA-interval
- Implement function $\Phi(\mathrm{SA}[i]) = \mathrm{SA}[i-1]$
- Can be implemented in $\mathcal{O}(r)$ space [6]

Illustration

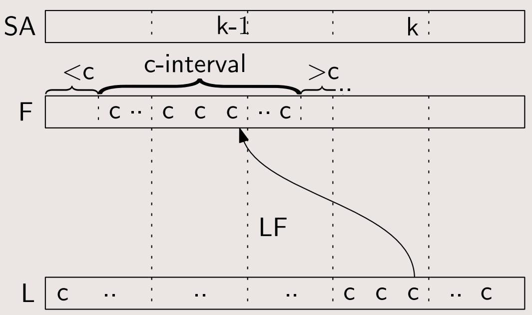


Compressed Text Indexes

LF in $O(r)$ space

- Fix a character c

Illustration

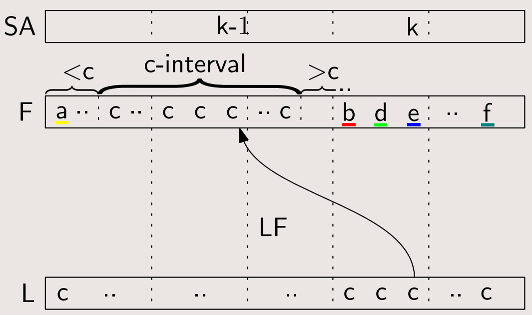


Compressed Text Indexes

LF in $O(r)$ space

- Fix a character c
- Rows i with $L[i] = c$ are sorted by what follows c in T

Illustration

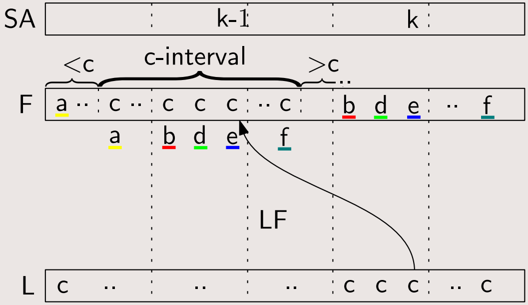


Compressed Text Indexes

LF in $O(r)$ space

- Fix a character c
- Rows i with $L[i] = c$ are sorted by what follows c in T
- Rows $LF(i)$ are also sorted by what follows c in T

Illustration



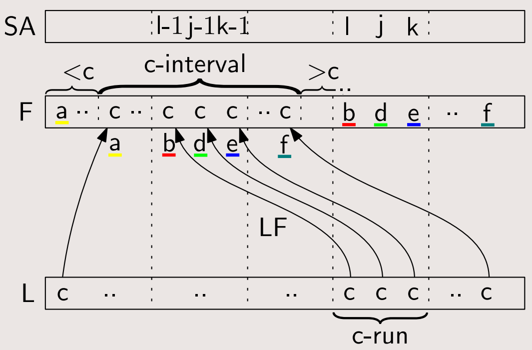
Compressed Text Indexes

LF in $O(r)$ space

- Fix a character c
- Rows i with $L[i] = c$ are sorted by what follows c in T
- Rows $LF(i)$ are also sorted by what follows c in T

$\Longrightarrow$ $LF(i)$ is ascending for a fixed $L[i] = c$

Illustration



Compressed Text Indexes

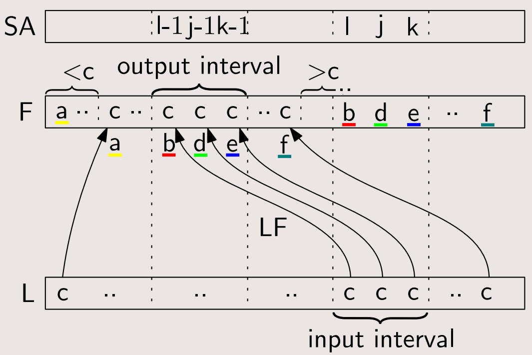
LF in $\mathcal{O}(r)$ space

- Fix a character c
- Rows i with $L[i] = c$ are sorted by what follows c in T
- Rows $LF(i)$ are also sorted by what follows c in T

$\Longrightarrow$ $LF(i)$ is ascending for a fixed $L[i] = c$

- Recall r = number of runs in L

Illustration



Compressed Text Indexes

LF in $O(r)$ space

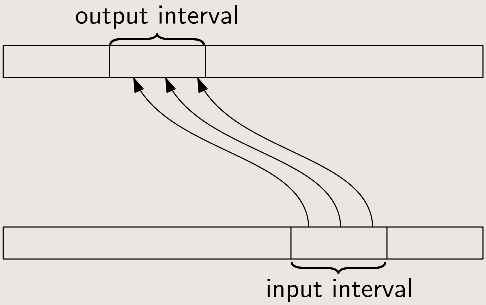
- Fix a character c
- Rows i with $L[i] = c$ are sorted by what follows c in T
- Rows $LF(i)$ are also sorted by what follows c in T

$\Longrightarrow$ $LF(i)$ is ascending for a fixed $L[i] = c$

- Recall r = number of runs in L

$\Longrightarrow$ LF can be divided into r intervals

Illustration



Compressed Text Indexes

LF in $O(r)$ space

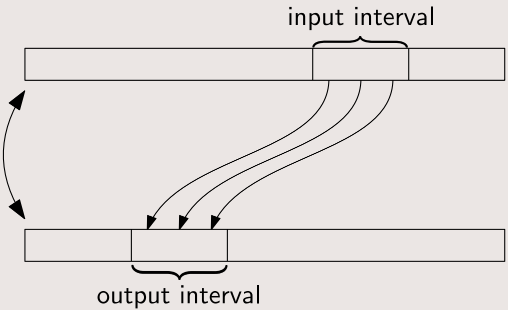
- Fix a character c
- Rows i with $L[i] = c$ are sorted by what follows c in T
- Rows $LF(i)$ are also sorted by what follows c in T

$\Longrightarrow$ $LF(i)$ is ascending for a fixed $L[i] = c$

- Recall r = number of runs in L

$\Longrightarrow$ LF can be divided into r intervals

Illustration



Compressed Text Indexes

LF in $O(r)$ space

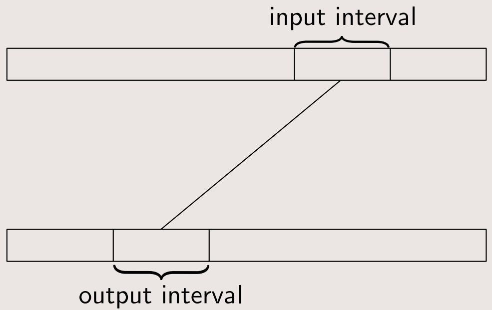
- Fix a character c
- Rows i with $\mathrm{L}[i] = c$ are sorted by what follows c in T
- Rows $\mathrm{LF}(i)$ are also sorted by what follows c in T

$\Longrightarrow$ $\mathrm{LF}(i)$ is ascending for a fixed $\mathrm{L}[i] = \mathrm{c}$

- Recall r = number of runs in L

$\Longrightarrow$ LF can be divided into r intervals

Illustration

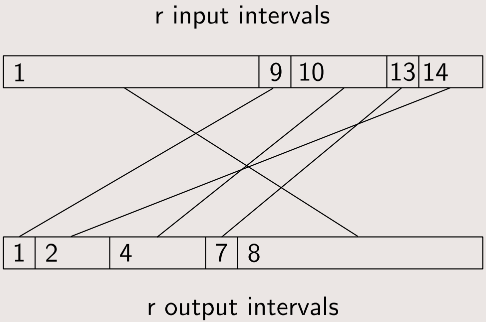


Compressed Text Indexes

Disjoint Interval Sequence

- $I = (p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k)$ with $d_i = p_{i+1} - p_i$ and $n + 1 = p_k + d_k$

Illustration

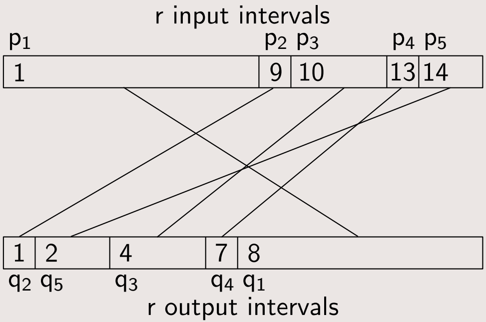


Compressed Text Indexes

Disjoint Interval Sequence

- $I = (p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k)$ with $d_i = p_{i+1} - p_i$ and $n + 1 = p_k + d_k$

Illustration

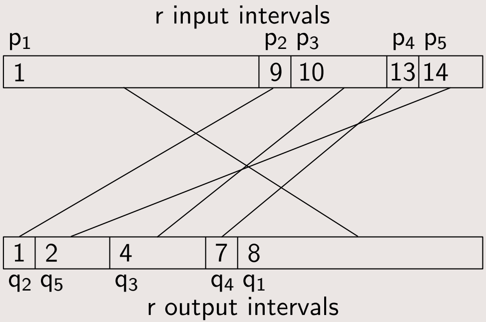


Compressed Text Indexes

Disjoint Interval Sequence

- $I = (p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k)$ with $d_i = p_{i+1} - p_i$ and $n + 1 = p_k + d_k$
- Input intervals $[p_i, p_i + d_i)$
- Corresponding output intervals $[q_i, q_i + d_i)$ have the same lengths d_i and do not overlap

Illustration



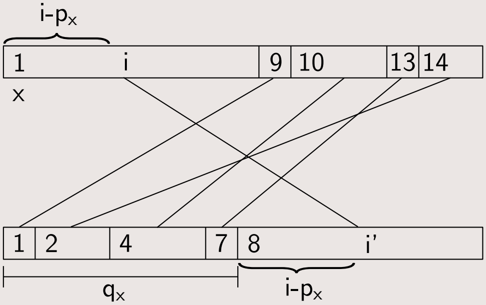
Compressed Text Indexes

Disjoint Interval Sequence

- $I = (p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k)$ with $d_i = p_{i+1} - p_i$ and $n + 1 = p_k + d_k$
- Input intervals $[p_i, p_i + d_i)$
- Corresponding output intervals $[q_i, q_i + d_i)$ have the same lengths d_i and do not overlap

$\Longrightarrow$ Represents function $f_I(i) = q_x + i - p_x$, where $i \in [p_x, p_x + d_x)$

Illustration

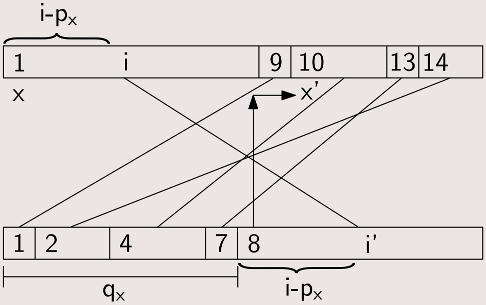


Compressed Text Indexes

Move Data Structure and Move Query

- $\mathrm{Move}(i, x) = (i', x')$ with $i' = f_I(i)$ and $i' \in [p_{x'}, p_{x'} + d_{x'})$

Illustration

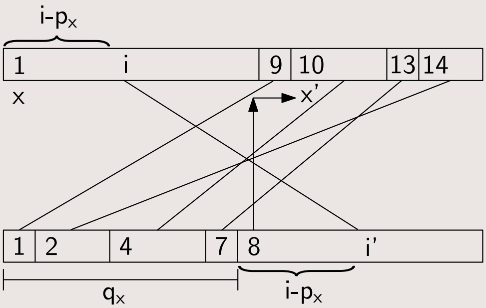


Compressed Text Indexes

Move Data Structure and Move Query

- Move(i, x) = (i', x') with $i' = f_I(i)$ and $i' \in [p_{x'}, p_{x'} + d_{x'})$
- Store $\mathrm{M_{idx}}[1..k]$, where $\mathrm{M_{idx}}[j]$ = index of the input interval containing q_j

Illustration



Compressed Text Indexes

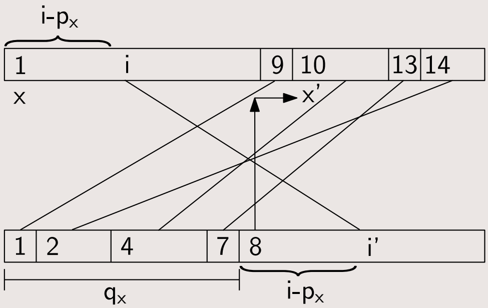
Move Data Structure and Move Query

- Move(i, x) = (i', x') with $i' = f_I(i)$ and $i' \in [p_{x'}, p_{x'} + d_{x'})$
- Store $\mathrm{M_{idx}}[1..k]$, where $\mathrm{M_{idx}}[j]$ = index of the input interval containing q_j

Move Query

- $\mathrm{M_{idx}} = [1, 1, 1, 1, 1]$

Illustration



Compressed Text Indexes

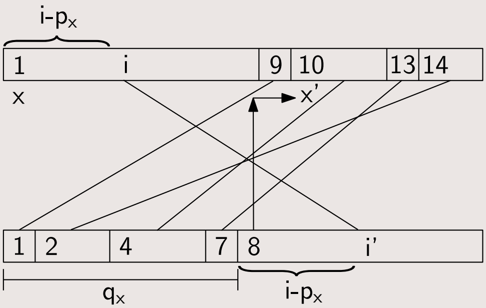
Move Data Structure and Move Query

- Move(i, x) = (i', x') with $i' = f_I(i)$ and $i' \in [p_{x'}, p_{x'} + d_{x'})$
- Store $\mathrm{M_{idx}}[1..k]$, where $\mathrm{M_{idx}}[j]$ = index of the input interval containing q_j

Move Query

- $\mathrm{M_{idx}} = [1, 1, 1, 1, 1]$
- Move(4, 1) = (12, 3)

Illustration



Compressed Text Indexes

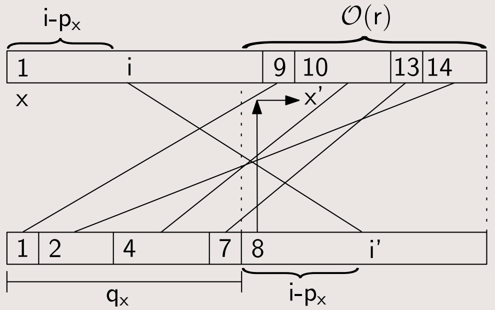
Move Data Structure and Move Query

- Move(i, x) = (i', x') with $i' = f_I(i)$ and $i' \in [p_{x'}, p_{x'} + d_{x'})$
- Store $\mathrm{M_{idx}}[1..k]$, where $\mathrm{M_{idx}}[j]$ = index of the input interval containing q_j
- Runtime $\mathcal{O}$(#input intervals starting in $[q_x, q_x + d_x)) = \mathcal{O}(r)$

Move Query

- $\mathrm{M_{idx}} = [1, 1, 1, 1, 1]$
- Move(4, 1) = (12, 3)

Illustration



Compressed Text Indexes

Move Data Structure and Move Query

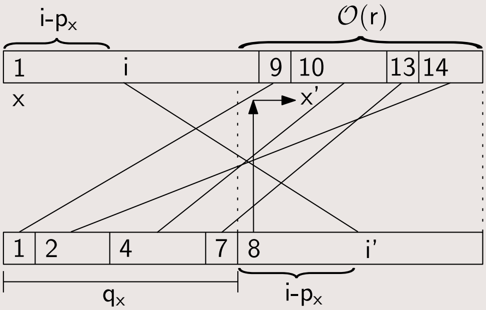
- Move$(i, x) = (i', x')$ with $i' = f_I(i)$ and $i' \in [p_{x'}, p_{x'} + d_{x'})$
- Store $\mathrm{M_{idx}}[1..k]$, where $\mathrm{M_{idx}}[j]$ = index of the input interval containing q_j
- Runtime $\mathcal{O}$(#input intervals starting in $[q_x, q_x + d_x)) = \mathcal{O}(r)$

$\Longrightarrow$ Limit to $\mathcal{O}(a)$ for $a \geq 2$

Move Query

- $\mathrm{M_{idx}} = [1, 1, 1, 1, 1]$
- Move$(4, 1) = (12, 3)$

Illustration

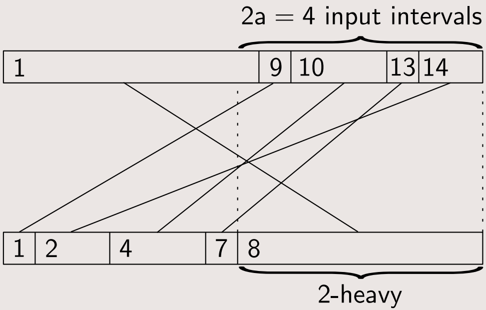


Compressed Text Indexes

a-balanced Disjoint Interval Sequence

- Output interval $[q_x, q_x + d_x)$ is a-heavy $\Leftrightarrow$ $\geq 2a$ input intervals start in $[q_x, q_x + d_x)$

Illustration

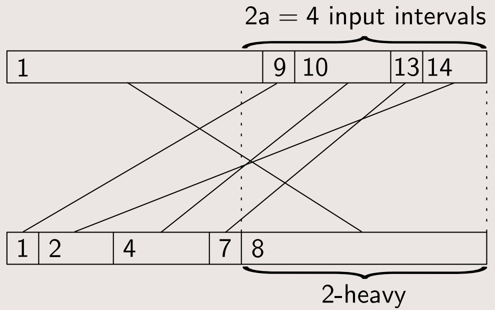


Compressed Text Indexes

a-balanced Disjoint Interval Sequence

- Output interval $[q_x, q_x + d_x)$ is a-heavy $\Leftrightarrow$ $\geq 2a$ input intervals start in $[q_x, q_x + d_x)$
- I is a-heavy $\Leftrightarrow$ there is an a-heavy output interval in I
- a-balanced $\Leftrightarrow$ not a-heavy

Illustration



Compressed Text Indexes

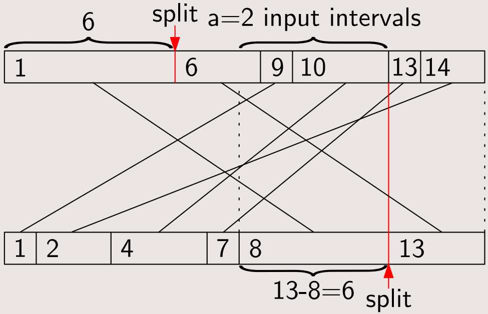
a-balanced Disjoint Interval Sequence

- Output interval $[q_x, q_x + d_x)$ is a-heavy $\Leftrightarrow$ $\geq 2a$ input intervals start in $[q_x, q_x + d_x)$
- I is a-heavy $\Leftrightarrow$ there is an a-heavy output interval in I
- a-balanced $\Leftrightarrow$ not a-heavy

Balancing Algorithm

- Iteratively splits a-heavy output intervals
- Terminates as soon as I is a-balanced

Illustration



Compressed Text Indexes

a-balanced Disjoint Interval Sequence

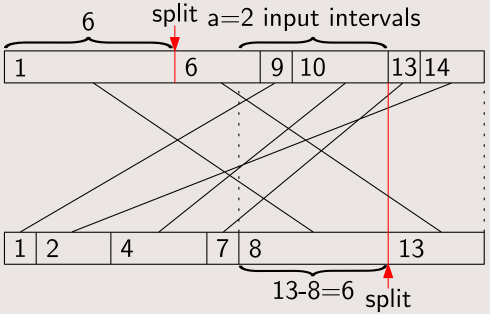
- Output interval $[q_x, q_x + d_x)$ is a-heavy $\Leftrightarrow$ $\geq 2a$ input intervals start in $[q_x, q_x + d_x)$
- I is a-heavy $\Leftrightarrow$ there is an a-heavy output interval in I
- a-balanced $\Leftrightarrow$ not a-heavy

Balancing Algorithm

- Iteratively splits a-heavy output intervals
- Terminates as soon as I is a-balanced
- Let t' = number of splits, and $k' = k + t'$

$\Rightarrow$ $k' \leq k\frac{a}{a-1} \leq 2k = O(k)$

Illustration

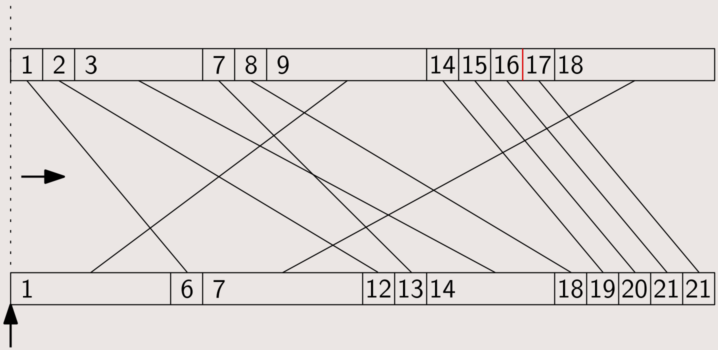


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals

Example Execution ($a = 2$)

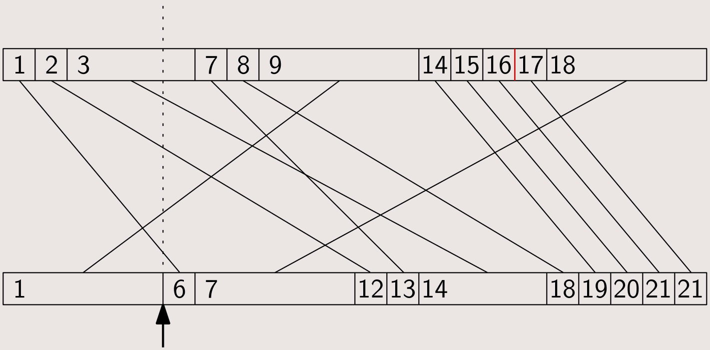


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals

Example Execution ($a = 2$)

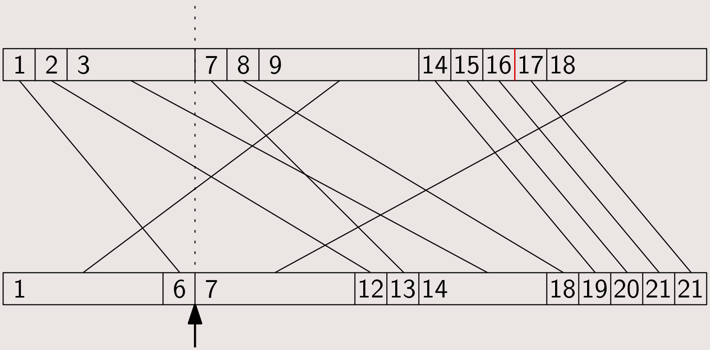


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals

Example Execution ($a = 2$)

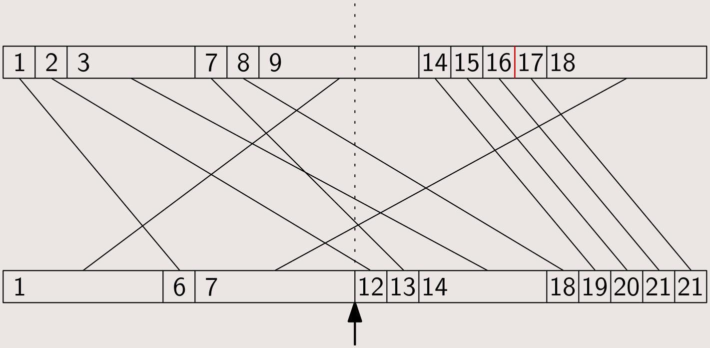


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals

Example Execution ($a = 2$)

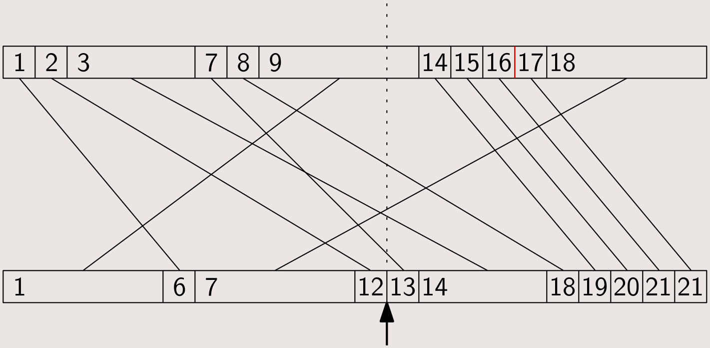


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals

Example Execution ($a = 2$)

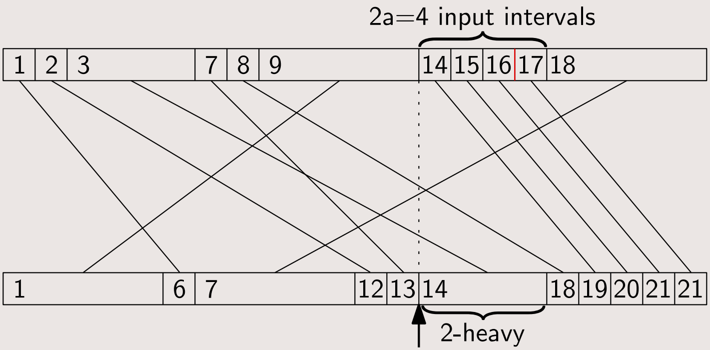


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:

Example Execution ($a = 2$)

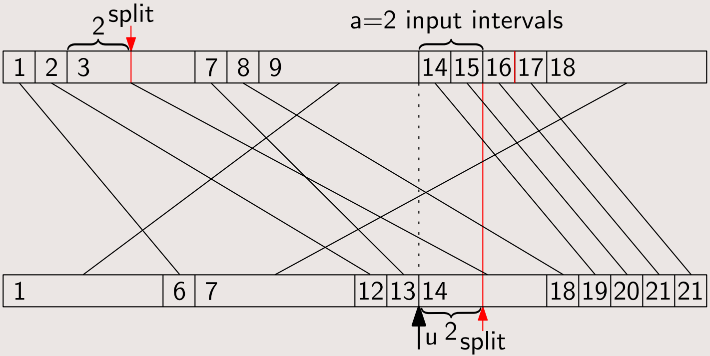


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u

Example Execution ($a = 2$)

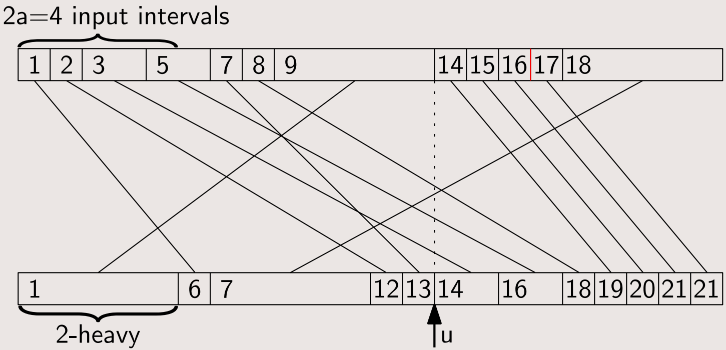


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
    - Split and remember starting position u
    - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)

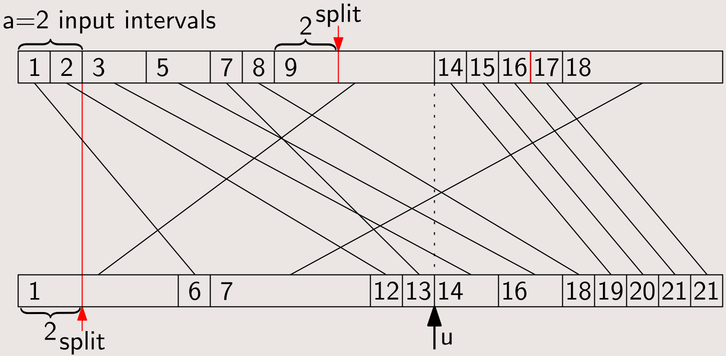


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)

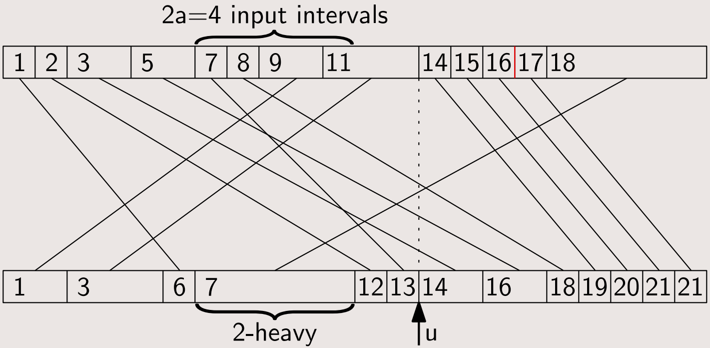


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)

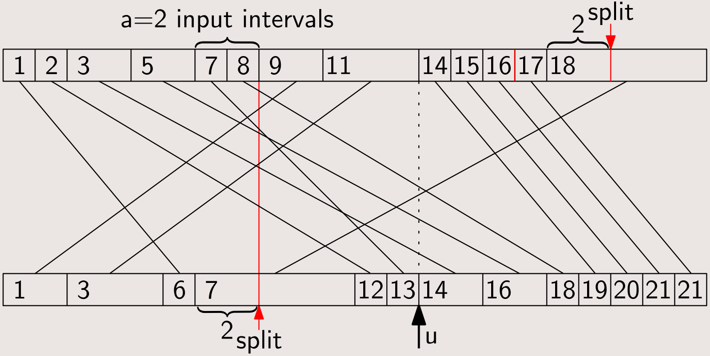


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)

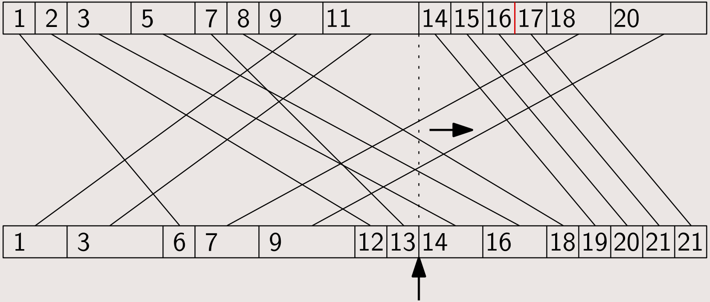


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)

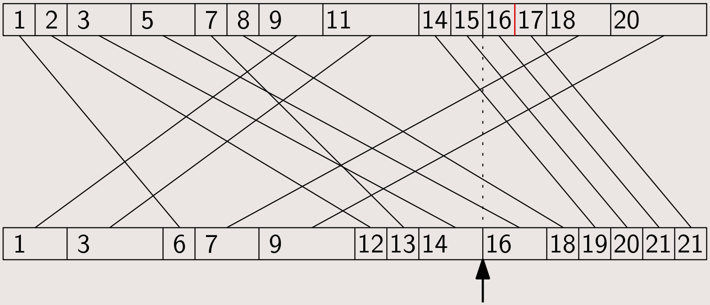


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)

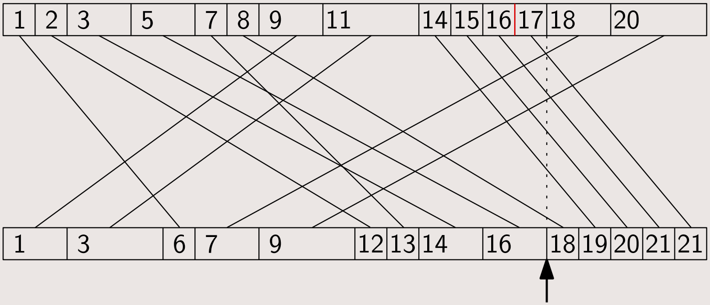


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)

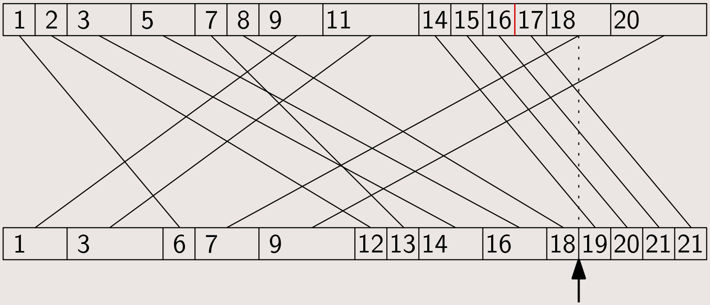


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)

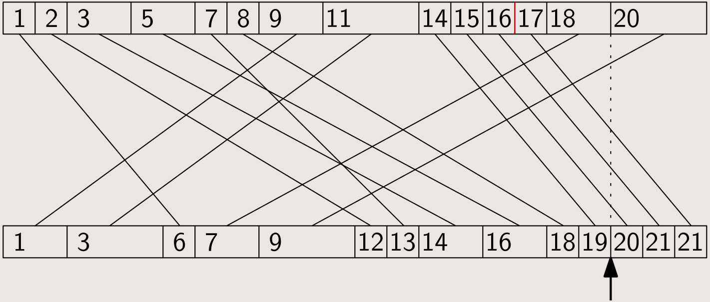


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)

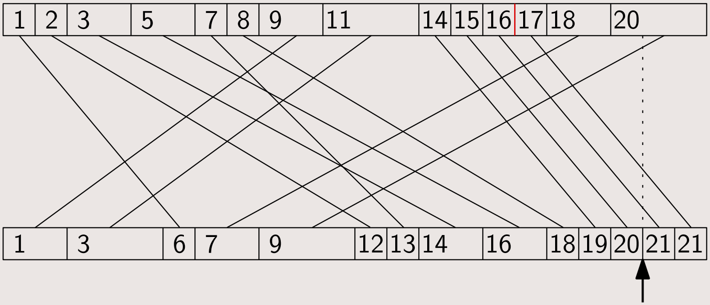


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)

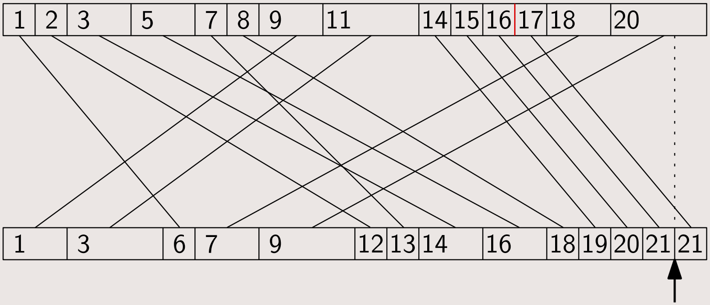


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
    - Split and remember starting position u
    - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)

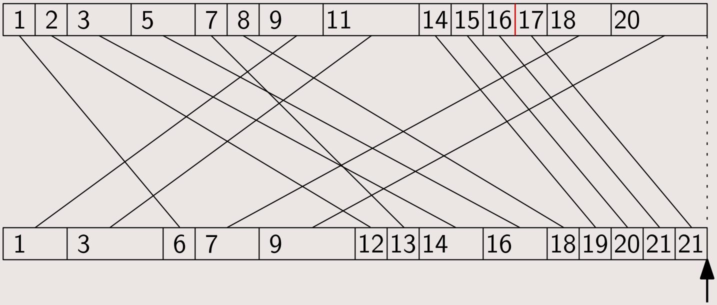


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Example Execution ($a = 2$)



Fast Balancing Algorithm

General Approach

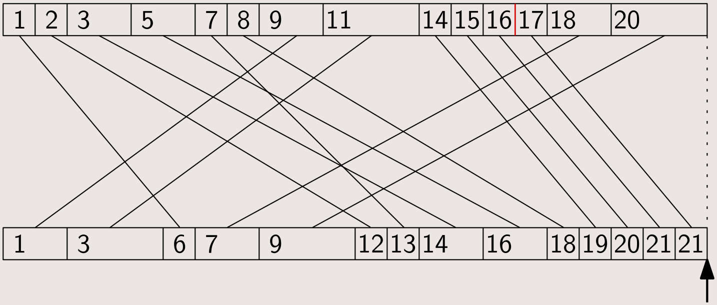
- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Details

- Use balanced search trees (B-Trees) for input and output intervals

$\Longrightarrow$ $\mathcal{O}(k \log k)$ time, $\mathcal{O}(1)$ space

Example Execution ($a = 2$)

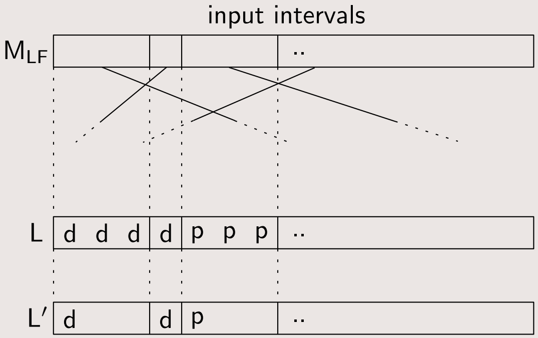


Move-r

Index Data Structures

- $|\mathrm{M}^{LF}| = r'$ move data structure for LF
- $L'[1..r']$ bwt characters of input intervals in M^{LF}
- $|RS_{L'}| = \mathcal{O}(r')$ rank-select data structure for L'

Backward Search



Move-r

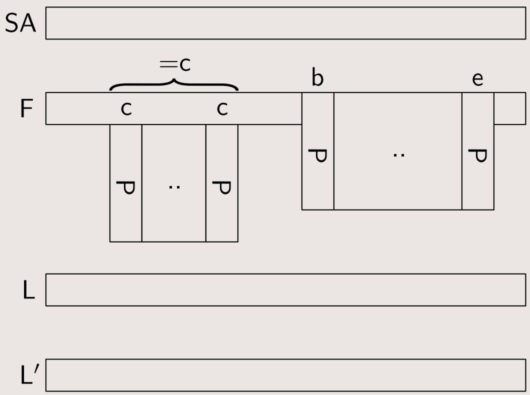
Index Data Structures

- $|\mathrm{M}^{LF}| = r'$ move data structure for LF
- $L'[1..r']$ bwt characters of input intervals in M^{LF}
- $|RS_{L'}| = \mathcal{O}(r')$ rank-select data structure for L'

Backward Search (Step)

- given SA-interval $[b, e]$ of P

Backward Search



Move-r

Index Data Structures

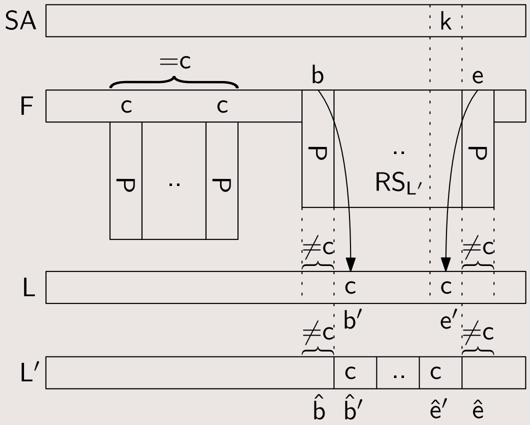
- $|\mathrm{M}^{LF}| = r'$ move data structure for LF
- $L'[1..r']$ bwt characters of input intervals in M^{LF}
- $|RS_{L'}| = \mathcal{O}(r')$ rank-select data structure for L'

Backward Search (Step)

- given SA-interval $[b, e]$ of P

1. compute b' and e' with L' and $RS_{L'}$ (maintain input interval indexes $\hat{b}, \hat{e}, \hat{b}', \hat{e}'$ of b, e, b', e')

Backward Search



Move-r

Index Data Structures

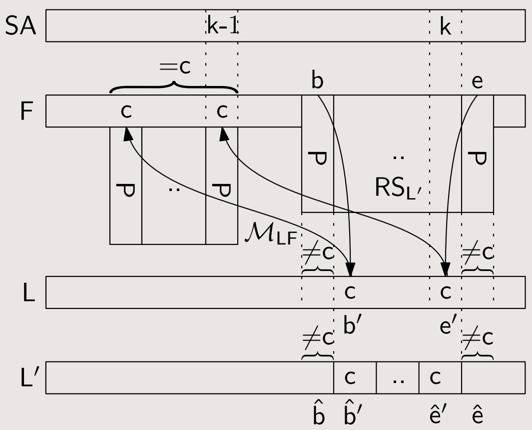
- $|\mathrm{M}^{LF}| = r'$ move data structure for LF
- $L'[1..r']$ bwt characters of input intervals in M^{LF}
- $|RS_{L'}| = \mathcal{O}(r')$ rank-select data structure for L'

Backward Search (Step)

- given SA-interval $[b, e]$ of P

1. compute b' and e' with L' and $RS_{L'}$ (maintain input interval indexes $\hat{b}, \hat{e}, \hat{b}', \hat{e}'$ of b, e, b', e')
2. compute SA-interval of cP with M^{LF}

Backward Search



Move-r

Index Data Structures

- $|\mathsf{M}^{LF}| = r'$ move data structure for LF
- $L'[1..r']$ bwt characters of input intervals in M^{LF}
- $|RS_{L'}| = \mathcal{O}(r')$ rank-select data structure for L'
- $|\mathsf{M}^{\Phi}| = r''$ move data structure for Φ
- $\mathsf{SA}_{\Phi}[1..r']$ (will be defined later)

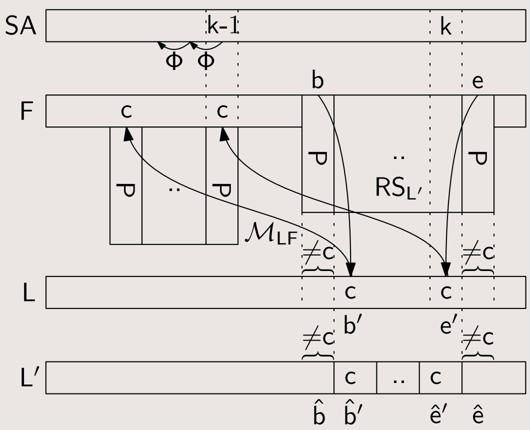
$\Big\}$ for locate

Backward Search (Step)

- given SA-interval $[b, e]$ of P

1. compute b' and e' with L' and $RS_{L'}$ (maintain input interval indexes $\hat{b}, \hat{e}, \hat{b}', \hat{e}'$ of b, e, b', e')
2. compute SA-interval of cP with M^{LF}

Backward Search

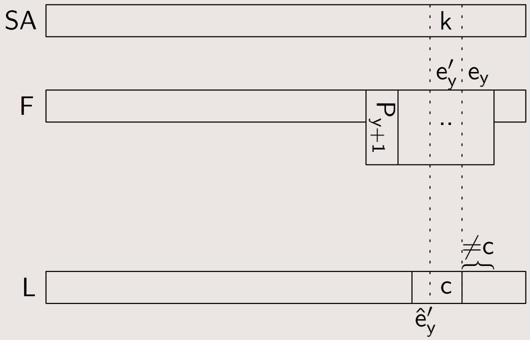


Optimized Locate Algorithm

Definitions

- e_i = end of SA-interval of P_{i+1}
- $e'_i = \mathrm{L.select}(P[i], \mathrm{L.rank}(P[i], e_i))$
- $\hat{e}'_i$ = index of M^{LF}-input interval containing e'_i

Backward Search

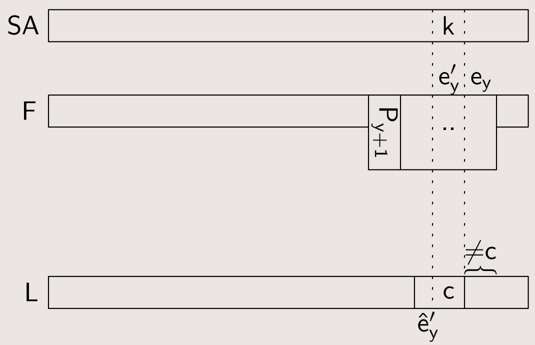


Optimized Locate Algorithm

Definitions

- e_i = end of SA-interval of P_{i+1}
- e'_i = L.select$(P[i]$, L.rank$(P[i], e_i))$
- $\hat{e}'_i$ = index of M^{LF}-input interval containing e'_i
- y = index of the last iteration in the backward search, where L$[e_i] \neq P[i]$ holds ($c = P[y]$)

Backward Search



Optimized Locate Algorithm

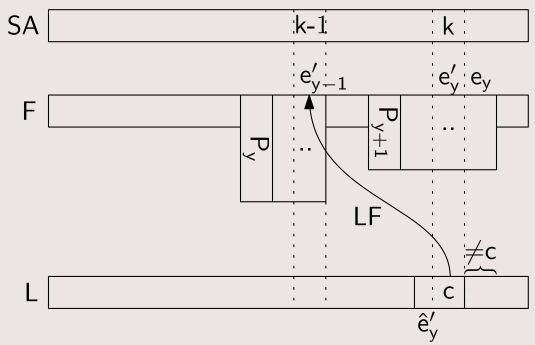
Definitions

- e_i = end of SA-interval of P_{i+1}
- $e'_i = \text{L.select}(P[i], \text{L.rank}(P[i], e_i))$
- $\hat{e}'_i$ = index of M^{LF}-input interval containing e'_i
- y = index of the last iteration in the backward search, where $\text{L}[e_i] \neq P[i]$ holds ($c = P[y]$)

Observation

- For $i \in [0, y)$, $\text{L}[e_i] = P[i]$ implies $e'_i = e_i = \text{LF}(e'_{i+1})$

Backward Search



Optimized Locate Algorithm

Definitions

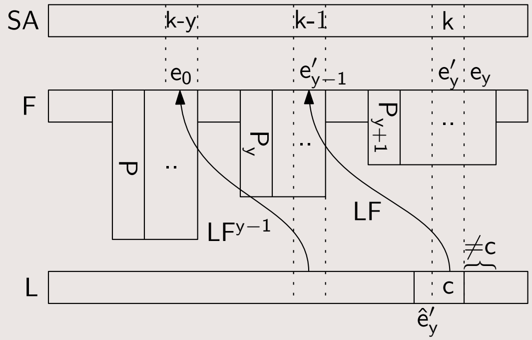
- e_i = end of SA-interval of P_{i+1}
- e'_i = L.select($P[i]$, L.rank($P[i], e_i$))
- $\hat{e}'_i$ = index of M^{LF}-input interval containing e'_i
- y = index of the last iteration in the backward search, where $\mathrm{L}[e_i] \neq P[i]$ holds ($c = P[y]$)

Observation

- For $i \in [0, y)$, $\mathrm{L}[e_i] = P[i]$ implies $e'_i = e_i = \mathrm{LF}(e'_{i+1})$

$\Longrightarrow$ By ind. $e_0 = \mathrm{LF}^y(e'_y) \Leftrightarrow \mathrm{SA}[e_0] = \mathrm{SA}[e'_y] - y$

Backward Search



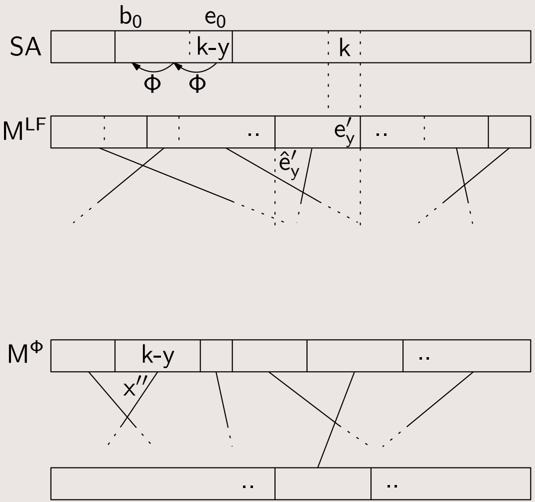
Optimized Locate Algorithm

General Procedure

- We want to compute $\mathrm{SA}[e_0] = k - y = \mathrm{SA}[e'_y] - y$ and the index x'' of the M^{Φ}-input interval containing $k - y$

$\Longrightarrow$ Then we can compute $\mathrm{SA}[b_0, e_0]$ with $e_0 - b_0$ Φ-move queries

Locate Phase



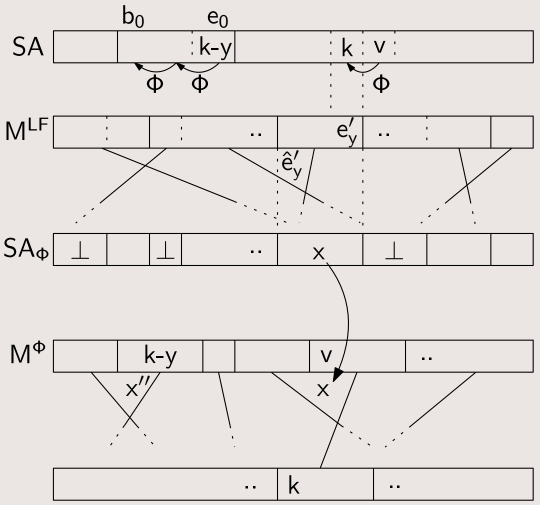
Optimized Locate Algorithm

Algorithm

- Obs.: there is an M^{Φ}-output interval starting with k

1. Compute index $x = SA_{\Phi}[\hat{e}'_y]$ of the M^{Φ}-input interval starting with v
2. Compute $k - y = M^{\Phi}_q[x] - y$

Locate Phase



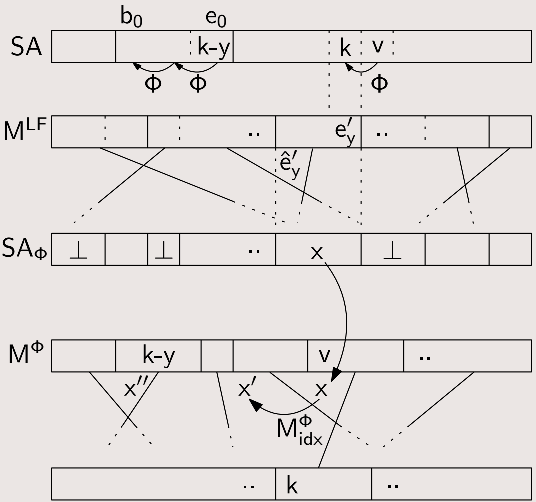
Optimized Locate Algorithm

Algorithm

- Obs.: there is an M^{Φ}-output interval starting with k

1. Compute index $x = SA_{\Phi}[\hat{e}'_y]$ of the M^{Φ}-input interval starting with v
2. Compute $k - y = M^{\Phi}_{q}[x] - y$
3. Compute index $x' = M^{\Phi}_{idx}[x]$ of the M^{Φ}-input interval containing k

Locate Phase



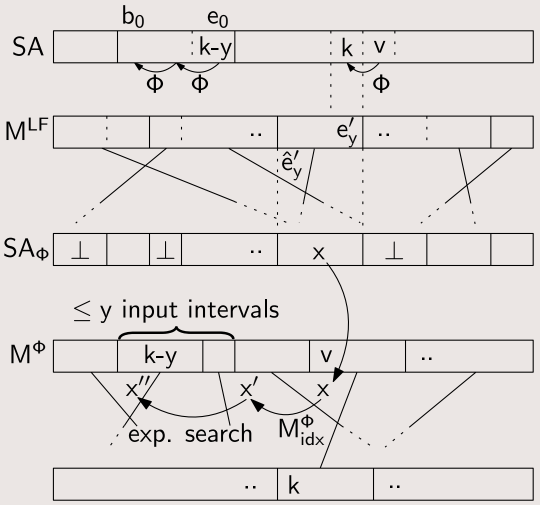
Optimized Locate Algorithm

Algorithm

- Obs.: there is an M^Φ-output interval starting with k

1. Compute index $x = SA_\Phi[\hat{e}'_y]$ of the M^Φ-input interval starting with v
2. Compute $k - y = M^\Phi_q[x] - y$
3. Compute index $x' = M^\Phi_{idx}[x]$ of the M^Φ-input interval containing k
4. Compute x'' with a $\mathcal{O}(\log y) = \mathcal{O}(\log m)$-time exponential search over the M^Φ-input intervals

Locate Phase



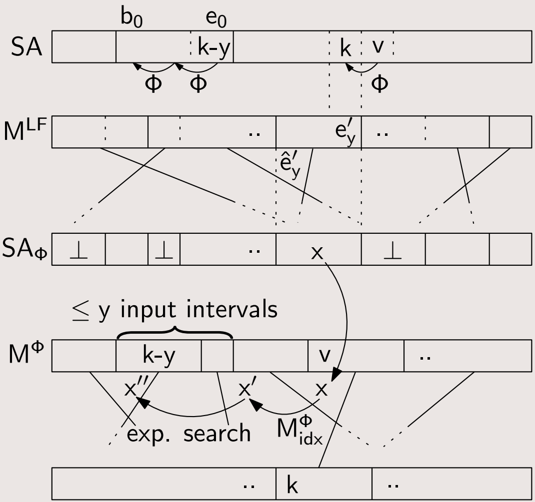
Optimized Locate Algorithm

Algorithm

- Obs.: there is an M^{Φ}-output interval starting with k

1. Compute index $x = SA_{\Phi}[\hat{e}'_y]$ of the M^{Φ}-input interval starting with v
2. Compute $k - y = M^{\Phi}_{q}[x] - y$
3. Compute index $x' = M^{\Phi}_{idx}[x]$ of the M^{Φ}-input interval containing k
4. Compute x'' with a $\mathcal{O}(\log y) = \mathcal{O}(\log m)$-time exponential search over the M^{Φ}-input intervals
5. Compute $SA[b_0, e_0]$ with $e_0 - b_0$ Φ-move queries

Locate Phase

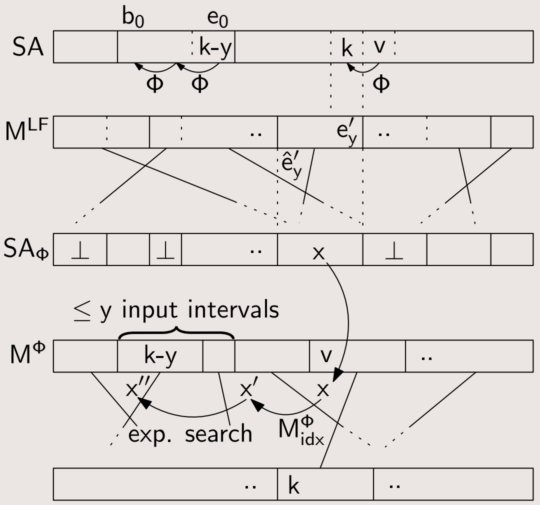


Optimized Locate Algorithm

Comparison with OptBWTR

- We can compute $\mathrm{SA}[e_0]$ and x'' in $\mathcal{O}(\log m)$ time and with $\mathcal{O}(\log m)$ cache misses
- OptBWTR: $\mathcal{O}(m)$ time and $3m$ cache misses

Locate Phase

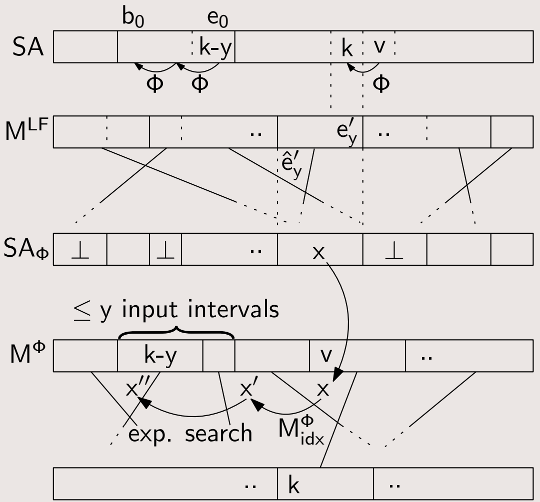


Optimized Locate Algorithm

Comparison with OptBWTR

- We can compute $SA[e_0]$ and x'' in $\mathcal{O}(\log m)$ time and with $\mathcal{O}(\log m)$ cache misses
- OptBWTR: $\mathcal{O}(m)$ time and $3m$ cache misses
- $SA_\Phi[1..r']$ requires $r'\lceil \log r'' \rceil$ bits
- OptBWTR: stores two arrays $SA^+[1..r']$ and $SA^+_{\text{index}}[1..r']$ ($r'(\lceil \log n \rceil + \lceil \log r'' \rceil)$ bits)

Locate Phase

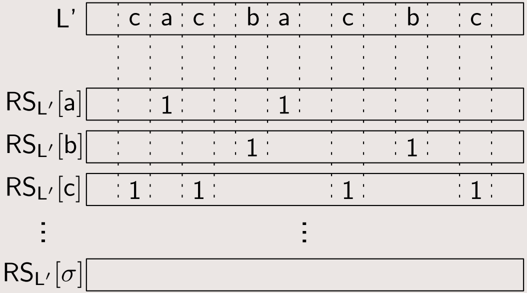


More Optimizations

Optimizations

- Faster rank-select data structure for $RS_{L'}$ using hybrid (compressed and uncompressed) bit-vectors with rank-select support

Illustration

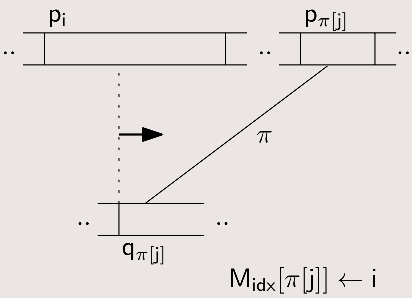


More Optimizations

Optimizations

- Faster rank-select data structure for $RS_{L'}$ using hybrid (compressed and uncompressed) bit-vectors with rank-select support
- Faster construction of $\mathrm{M_{idx}}$ and SA_Φ with perm. π of $[1, k']$, s.t. $q_{\pi[1]} < q_{\pi[2]} < \ldots < q_{\pi[k']}$

Illustration

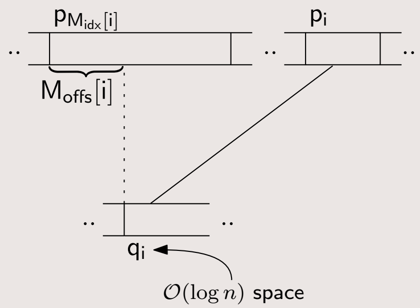


More Optimizations

Optimizations

- Faster rank-select data structure for $RS_{L'}$ using hybrid (compressed and uncompressed) bit-vectors with rank-select support
- Faster construction of $\mathsf{M}_{\mathsf{idx}}$ and SA_{Φ} with perm. π of $[1, k']$, s.t. $q_{\pi[1]} < q_{\pi[2]} < \ldots < q_{\pi[k']}$
- Smaller representation of the Move Data Structure (store $\mathsf{M}_{\mathsf{offs}}[1..k']$ instead of $\mathsf{M}_{\mathsf{q}}[1..k']$)

Illustration

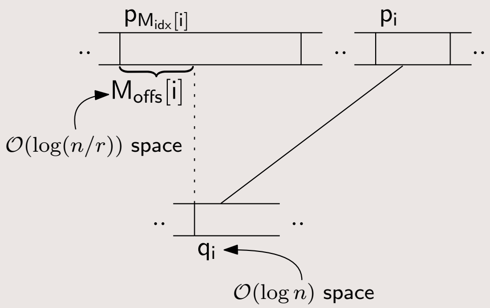


More Optimizations

Optimizations

- Faster rank-select data structure for $RS_{L'}$ using hybrid (compressed and uncompressed) bit-vectors with rank-select support
- Faster construction of $\mathsf{M}_{\mathsf{idx}}$ and SA_{Φ} with perm. π of $[1, k']$, s.t. $q_{\pi[1]} < q_{\pi[2]} < \ldots < q_{\pi[k']}$
- Smaller representation of the Move Data Structure (store $\mathsf{M}_{\mathsf{offs}}[1..k']$ instead of $\mathsf{M}_{\mathsf{q}}[1..k']$)

Illustration

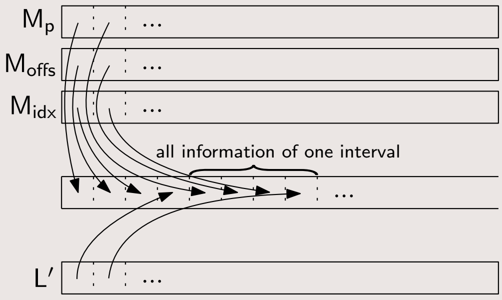


More Optimizations

Optimizations

- Faster rank-select data structure for $RS_{L'}$ using hybrid (compressed and uncompressed) bit-vectors with rank-select support
- Faster construction of $\mathrm{M_{idx}}$ and SA_Φ with perm. π of $[1, k']$, s.t. $q_{\pi[1]} < q_{\pi[2]} < \ldots < q_{\pi[k']}$
- Smaller representation of the Move Data Structure (store $\mathrm{M_{offs}}[1..k']$ instead of $\mathrm{M_q}[1..k']$)
- Reducing cache misses caused by array-lookups:
  - Interleaving arrays where possible

Illustration

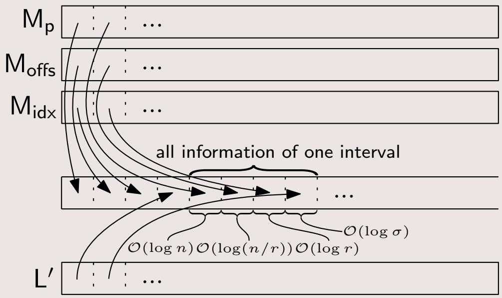


More Optimizations

Optimizations

- Faster rank-select data structure for $RS_{L'}$ using hybrid (compressed and uncompressed) bit-vectors with rank-select support
- Faster construction of $\mathsf{M}_{\mathsf{idx}}$ and SA_Φ with perm. π of $[1, k']$, s.t. $q_{\pi[1]} < q_{\pi[2]} < \ldots < q_{\pi[k']}$
- Smaller representation of the Move Data Structure (store $\mathsf{M}_{\mathsf{offs}}[1..k']$ instead of $\mathsf{M}_{\mathsf{q}}[1..k']$)
- Reducing cache misses caused by array-lookups:
  - Interleaving arrays where possible
  - Variable word-width byte-aligned arrays (use the minimum necessary number of bytes per entry)

Illustration

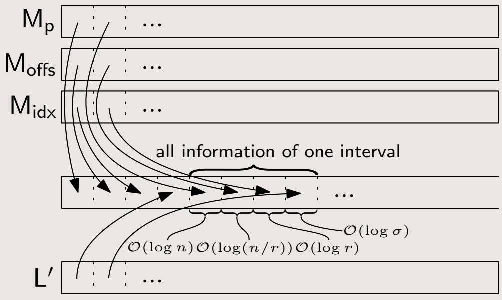


More Optimizations

Optimizations

- Faster rank-select data structure for $RS_{L'}$ using hybrid (compressed and uncompressed) bit-vectors with rank-select support
- Faster construction of $\mathrm{M_{idx}}$ and SA_Φ with perm. π of $[1, k']$, s.t. $q_{\pi[1]} < q_{\pi[2]} < \ldots < q_{\pi[k']}$
- Smaller representation of the Move Data Structure (store $\mathrm{M_{offs}}[1..k']$ instead of $\mathrm{M_q}[1..k']$)
- Reducing cache misses caused by array-lookups:
  - Interleaving arrays where possible
  - Variable word-width byte-aligned arrays (use the minimum necessary number of bytes per entry)
- ...

Illustration



Experimental Setup

Tested Indexes

- ◆ move-r (static, Big-BWT [2])
- ■ r-index [6] (static, Big-BWT [2])
- ▲ online-rlbwt [1] (dynamic)
- ● rcomp-glfig [7] (dynamic)

Experimental Setup

Tested Indexes

- ◆ move-r (static, Big-BWT [2])
- ■ r-index [6] (static, Big-BWT [2])
- ▲ online-rlbwt [1] (dynamic)
- ● rcomp-glfig [7] (dynamic)
- ∗ r-index-f [3] (static, Big-BWT [2])
- ⅄ block-rlbwt-2 [5] (static, grlBWT [4])
- + block-rlbwt-v [5] (static, grlBWT [4])
- × block-rlbwt-r [5] (static, grlBWT [4])

only count

Experimental Setup

Tested Indexes

- ◆ move-r (static, Big-BWT [2])
- ■ r-index [6] (static, Big-BWT [2])
- ▲ online-rlbwt [1] (dynamic)
- ● rcomp-glfig [7] (dynamic)
- ∗ r-index-f [3] (static, Big-BWT [2])
- ⅄ block-rlbwt-2 [5] (static, grlBWT [4])
- \+ block-rlbwt-v [5] (static, grlBWT [4])
- × block-rlbwt-r [5] (static, grlBWT [4])

only count

Measured Texts

text	size [GB]	σ	n/r	r'/r	r''/r
einstein.en.txt	0.47	139	1611.18	1.23	1.49
sars2	84.19	80	686.57	1.38	1.06
dewiki	68.72	210	345.80	1.23	1.35
chr19	58.57	52	272.20	1.06	2.00
english	2.21	239	3.36	1.19	1.20

More Information

- $n/r \approx$ compressibility
- σ = alphabet size
- r', r'' = number of intervals in M^{LF}/M^{Φ} (for $a = 2$)

Experimental Setup

Tested Indexes

- ◆ move-r (static, Big-BWT [2])
- ■ r-index [6] (static, Big-BWT [2])
- ▲ online-rlbwt [1] (dynamic)
- ● rcomp-glfig [7] (dynamic)
- ∗ r-index-f [3] (static, Big-BWT [2])
- ⅄ block-rlbwt-2 [5] (static, grlBWT [4])
- + block-rlbwt-v [5] (static, grlBWT [4])
- × block-rlbwt-r [5] (static, grlBWT [4])

only count

Measured Texts

text	size [GB]	σ	n/r	r'/r	r''/r
einstein.en.txt	0.47	139	1611.18	1.23	1.49
sars2	84.19	80	686.57	1.38	1.06
dewiki	68.72	210	345.80	1.23	1.35
chr19	58.57	52	272.20	1.06	2.00
english	2.21	239	3.36	1.19	1.20

More Information

- $n/r \approx$ compressibility
- σ = alphabet size
- r', r'' = number of intervals in M^{LF}/M^{Φ} (for $a = 2$)
- $a = 8$ is the optimal trade-off between space and query throughput

$\implies$ The following measurements use $a = 8$

Experimental Setup

Tested Indexes

- ◆ move-r (static, Big-BWT [2])
- ■ r-index [6] (static, Big-BWT [2])
- ▲ online-rlbwt [1] (dynamic)
- ● rcomp-glfig [7] (dynamic)
- ∗ r-index-f [3] (static, Big-BWT [2])
- ⅄ block-rlbwt-2 [5] (static, grlBWT [4])
- + block-rlbwt-v [5] (static, grlBWT [4])
- × block-rlbwt-r [5] (static, grlBWT [4])

$\left.\begin{matrix}\text{r-index-f}\\ \text{block-rlbwt-2}\\ \text{block-rlbwt-v}\\ \text{block-rlbwt-r}\end{matrix}\right\}$ only count

Measured Texts

text	size [GB]	σ	n/r	r'/r	r''/r
einstein.en.txt	0.47	139	1611.18	1.23	1.49
sars2	84.19	80	686.57	1.38	1.06
dewiki	68.72	210	345.80	1.23	1.35
chr19	58.57	52	272.20	1.06	2.00
english	2.21	239	3.36	1.19	1.20

Test System

- 2x AMD EPYC 7452 (32/64x 2.35-3.35GHz, 2/16/128MB L1/2/3 cache)
- 1TB 3200 MT/s DDR4 RAM
- GCC 9.4.0 with flags `"-march=native -DNDEBUG -Ofast"`
- Ubuntu 18.04.6

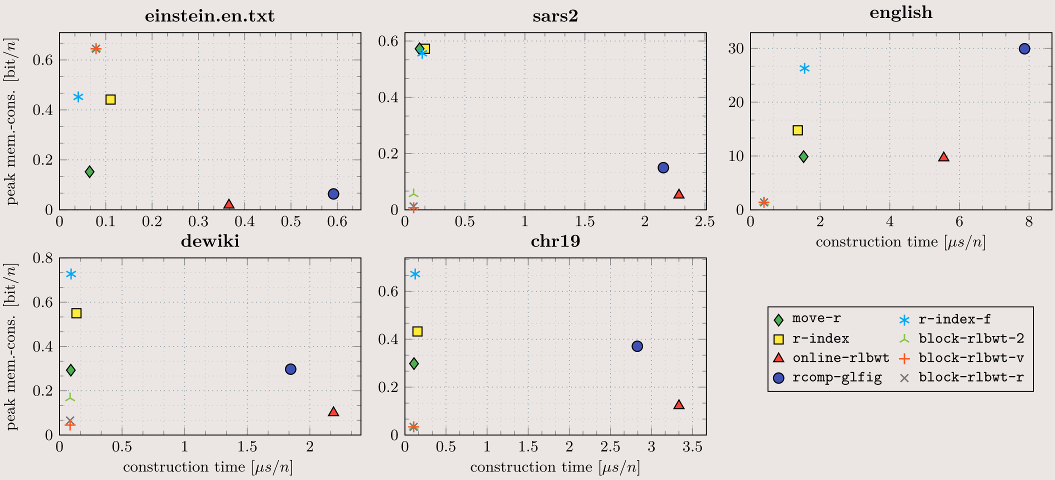
More Information

- $n/r \approx$ compressibility
- σ = alphabet size
- r', r'' = number of intervals in M^{LF}/M^{Φ} (for $a = 2$)
- $a = 8$ is the optimal trade-off between space and query throughput

$\Longrightarrow$ The following measurements use $a = 8$

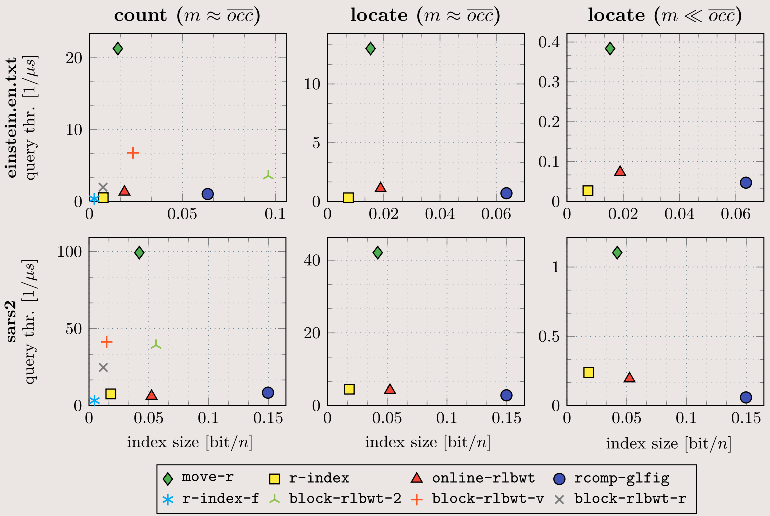
Results

Construction Performance



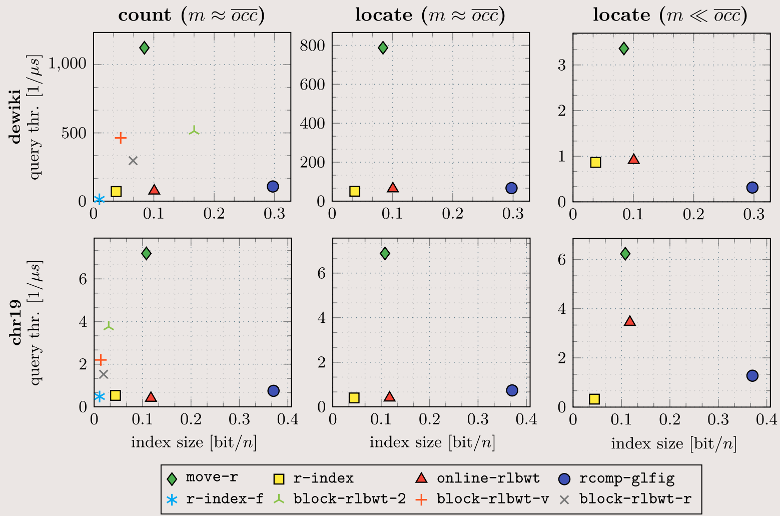
Results

Query Performance



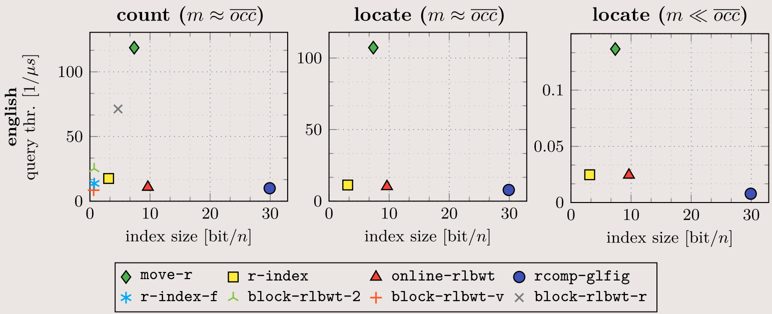
Results

Query Performance



Results

Query Performance



Summary

- 2x-35x (typ. 15x) faster queries
- 0.8x-2.5x (typ. 2x) larger index
- 0.9-2x (typ. 2x) faster construction with 1-3x (typ. 2x) lower memory usage

Bibliography

[1] Hideo Bannai, Travis Gagie, and Tomohiro I. "Refining the r-index". In: *Theoretical Computer Science* 812 (2020), pp. 96–108.

[2] Christina Boucher et al. "Prefix-Free Parsing for Building Big BWTs". In: *18th International Workshop on Algorithms in Bioinformatics WABI*. 2018, 2:1–2:16.

[3] Nathaniel K. Brown, Travis Gagie, and Massimiliano Rossi. "RLBWT Tricks". In: *20th International Symposium on Experimental Algorithms SEA*. 2022, 16:1–16:16.

[4] Diego Díaz-Domínguez and Gonzalo Navarro. "Efficient Construction of the BWT for Repetitive Text Using String Compression". In: *33rd Annual Symposium on Combinatorial Pattern Matching CPM*. 2022, 29:1–29:18.

[5] Diego Díaz-Domínguez et al. "Simple Runs-Bounded FM-Index Designs Are Fast". In: *21st International Symposium on Experimental Algorithms SEA*. 2023, 7:1–7:16.

[6] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. "Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space". In: *Journal of the ACM* 67.1 (2020), 2:1–2:54.

[7] Takaaki Nishimoto, Shunsuke Kanda, and Yasuo Tabei. "An Optimal-Time RLBWT Construction in BWT-Runs Bounded Space". In: *49th International Colloquium on Automata, Languages, and Programming ICALP*. 2022, 99:1–99:20.

[8] Takaaki Nishimoto and Yasuo Tabei. "Optimal-Time Queries on BWT-Runs Compressed Indexes". In: *48th International Colloquium on Automata, Languages, and Programming ICALP*. 2021, 101:1–101:15.