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

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

Queries

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

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**$(ac) = \{1, 6\}$

BWT-Based Text Indexes

BWT-based Text Indexes

- Text T of length n
- Lexicographical order $\prec$ (i.e. $abc \prec acb$)

BWT-Based Text Indexes

BWT-based Text Indexes

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

Burrows Wheeler Matrix (BWM)

- $T = acbcbac\$$

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

BWT-Based Text Indexes

BWT-based Text Indexes

- Text T of length n
- Lexicographical order $\prec$ (i.e. $abc \prec acb$)
- Suffix $T_i = T[i, n]$
- 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

BWT-Based Text Indexes

BWT-based Text Indexes

- Text T of length n
- Lexicographical order $\prec$ (i.e. $abc \prec acb$)
- Suffix $T_i = T[i, n]$
- 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

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

BWT-Based Text Indexes

BWT-based Text Indexes

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

SA-Interval

- $P = \underline{ac}$ has SA-interval $[2, 3]$

$\Rightarrow$ $\mathbf{count}(P) = |[2, 3]| = 2$

$\Rightarrow$ $\mathbf{locate}(P) = \mathsf{SA}[2, 3] = \{6, 1\}$

Burrows Wheeler Matrix (BWM)

- $T = \underline{a}cbcb\underline{a}c\$$
- i-th row = $\mathsf{rot}(T, \mathsf{SA}[i])$

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

Compressed BWT-based Text Indexes

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 [Gagie et al. 2020]:
  - $\mathcal{O}(r)$ 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

Compressed BWT-based Text Indexes

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 [Gagie et al. 2020]:
  - $\mathcal{O}(r)$ 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
- OptBWTR [Nishimoto et al. 2021] (not yet implemented):
  - $\mathcal{O}(r)$ space
  - Implements functions LF and Φ in $\underline{\mathcal{O}(1)\text{ time}}$ using <u>move data structures</u>
  - Count: $\mathcal{O}(m \log\log_\omega \sigma)$ time
  - Locate: additional $\mathcal{O}(occ)$ time

Compressed BWT-based Text Indexes

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 [Gagie et al. 2020]:
  - $\mathcal{O}(r)$ 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 [Nishimoto et al. 2021] (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 runtime reflected in practice?

Our Contribution

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

Our Contribution

Our Contribution

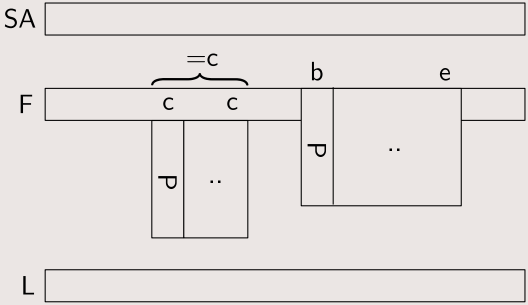
- Move-r: practically optimized implementation of OptBWTR
  - Practically optimized implementation and construction of the move data structure and other index data structures
  - Practically optimized count- and locate algorithms
  - More optimizations
- Compared with the resp. fastest other index:
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

BWT-Runs Compressed Text Indexes

Backward Search (Step)

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

Illustration

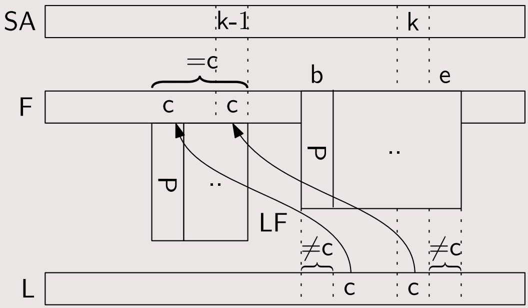


BWT-Runs Compressed Text Indexes

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



BWT-Runs Compressed Text Indexes

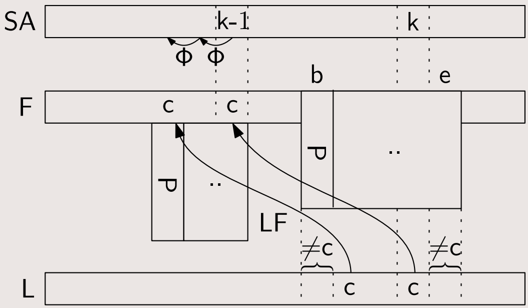
Backward Search (Step)

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

Locate Query

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

Illustration

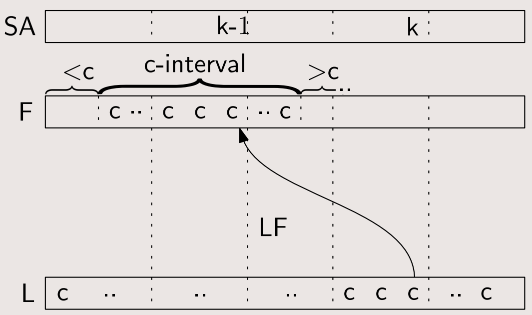


BWT-Runs Compressed Text Indexes

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

- Fix a character c

Illustration

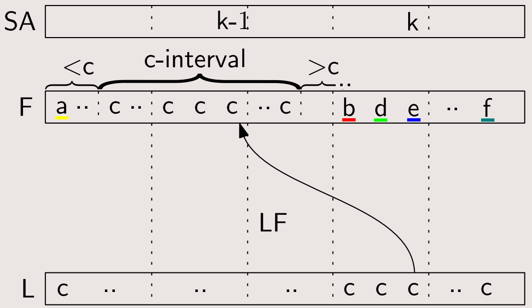


BWT-Runs Compressed Text Indexes

LF in $O(r)$ space

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

Illustration

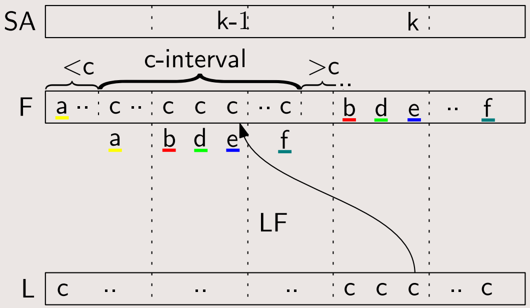


BWT-Runs Compressed Text Indexes

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

Illustration



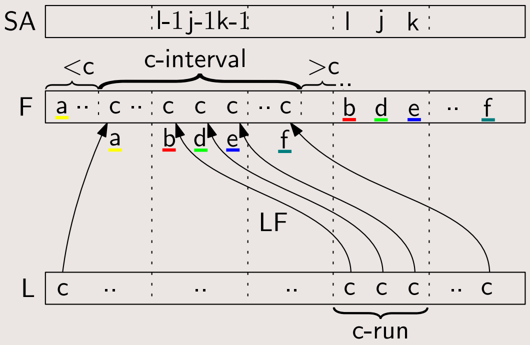
BWT-Runs Compressed Text Indexes

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 $L[i] = c$

Illustration



BWT-Runs Compressed Text Indexes

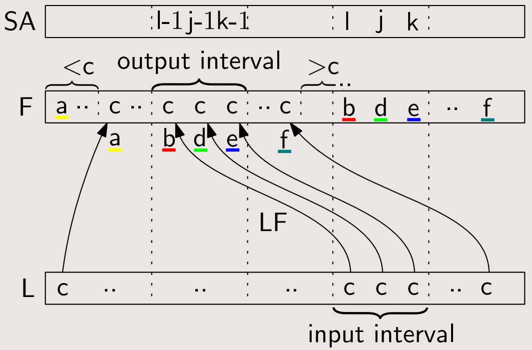
LF in $\mathcal{O}(r)$ space

- 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



BWT-Runs Compressed Text Indexes

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

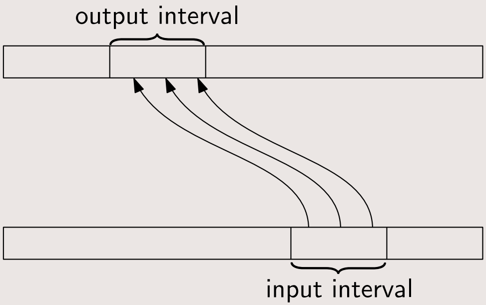
- Fix a character c
- Rows i with $L[i] = c$ are sorted by what follows c in T
- Rows $LF(i)$ are also sorted by what follows c in T

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

- Recall r = number of runs in L

$\Longrightarrow$ LF can be divided into r intervals

Illustration



BWT-Runs Compressed Text Indexes

LF in $O(r)$ space

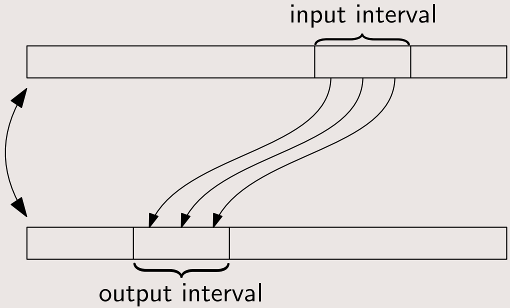
- Fix a character c
- Rows i with $\mathrm{L}[i] = c$ are sorted by what follows c in T
- Rows $\mathrm{LF}(i)$ are also sorted by what follows c in T

$\Longrightarrow$ $\mathrm{LF}(i)$ 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



BWT-Runs Compressed Text Indexes

LF in $O(r)$ space

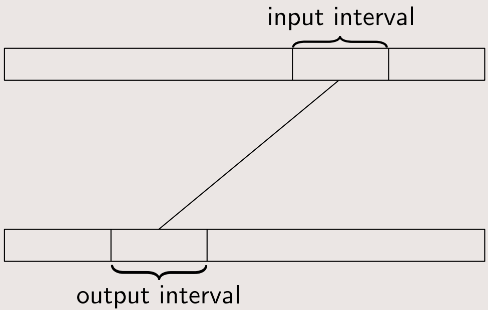
- Fix a character c
- Rows i with $L[i] = c$ are sorted by what follows c in T
- Rows $LF(i)$ are also sorted by what follows c in T

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

- Recall r = number of runs in L

$\Longrightarrow$ LF can be divided into r intervals

Illustration



BWT-Runs Compressed Text Indexes

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

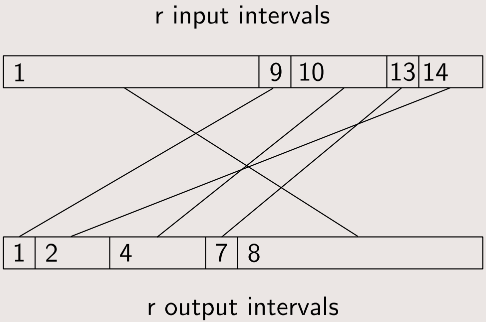
- Fix a character c
- Rows i with $L[i] = c$ are sorted by what follows c in T
- Rows $\mathrm{LF}(i)$ are also sorted by what follows c in T

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

- Recall r = number of runs in L

$\Longrightarrow$ LF can be divided into r intervals

Illustration

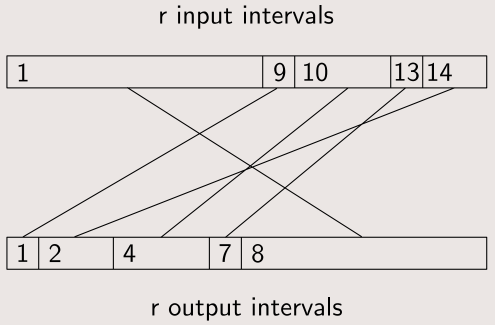


Move Data Structure

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

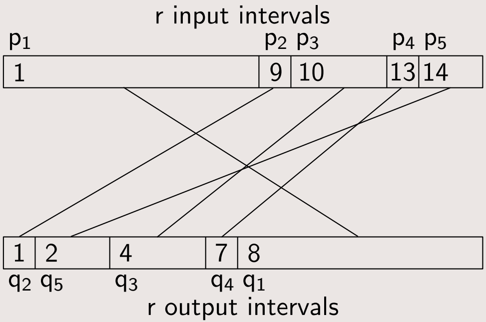


Move Data Structure

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

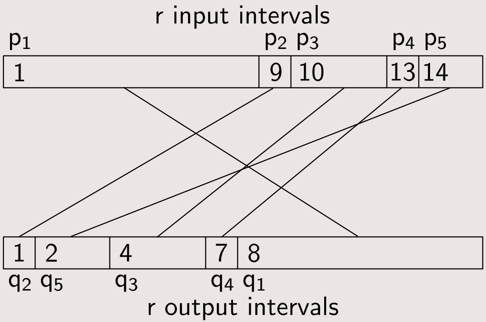


Move Data Structure

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



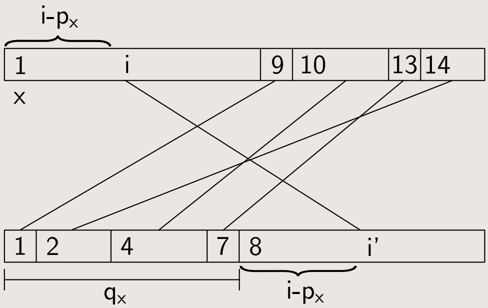
Move Data Structure

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

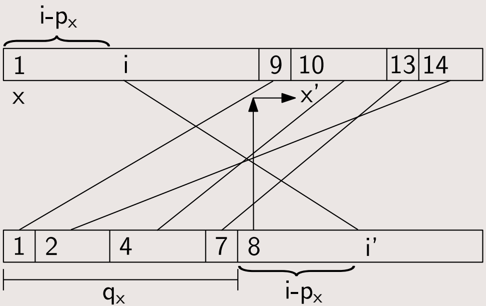


Move Data Structure

Move Data Structure and Move Query

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

Illustration

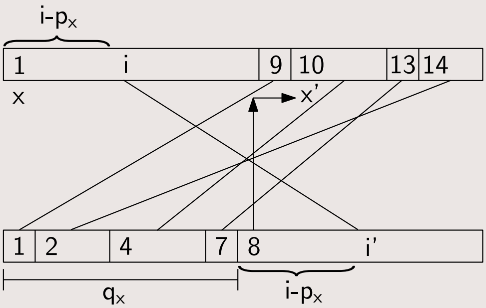


Move Data Structure

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



Move Data Structure

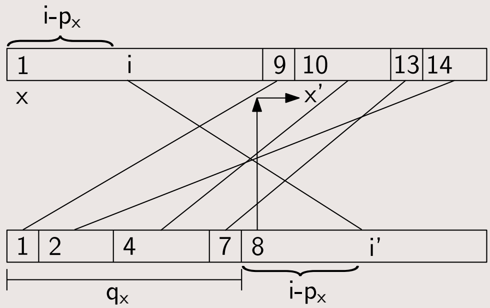
Move Data Structure and Move Query

- 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



Move Data Structure

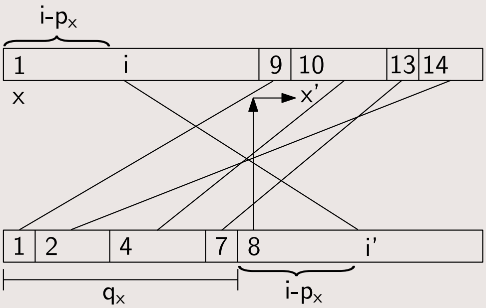
Move Data Structure and Move Query

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

Move Query

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

Illustration



Move Data Structure

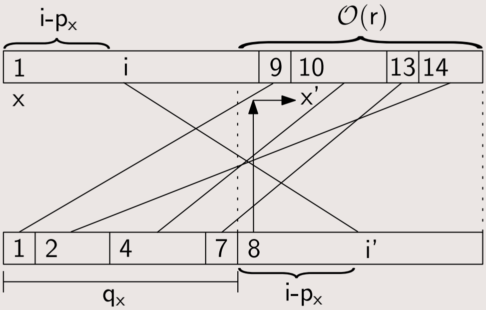
Move Data Structure and Move Query

- 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



Move Data Structure

Move Data Structure and Move Query

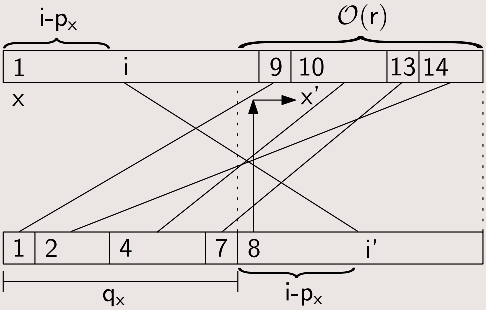
- Move$(i, x) = (i', x')$ with $i' = f_I(i)$ and $i' \in [p_{x'}, p_{x'} + d_{x'})$
- Store $\mathrm{M_{idx}}[1..k]$, where $\mathrm{M_{idx}}[j]$ = index of the input interval containing q_j
- Runtime $\mathcal{O}$(#input intervals starting in $[q_x, q_x + d_x)) = \mathcal{O}(r)$

$\Longrightarrow$ can be limited to $\mathcal{O}(1)$

Move Query

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

Illustration

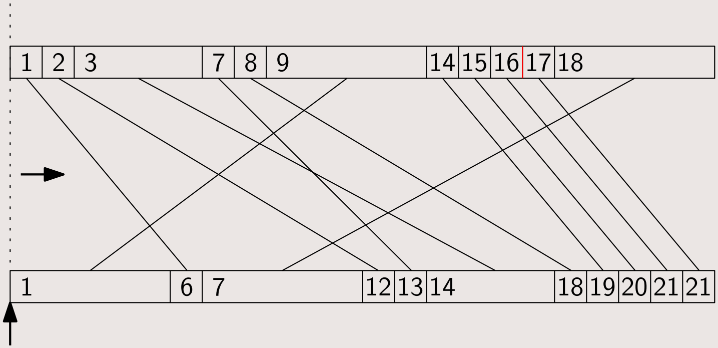


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals

Example Execution ($a = 2$)

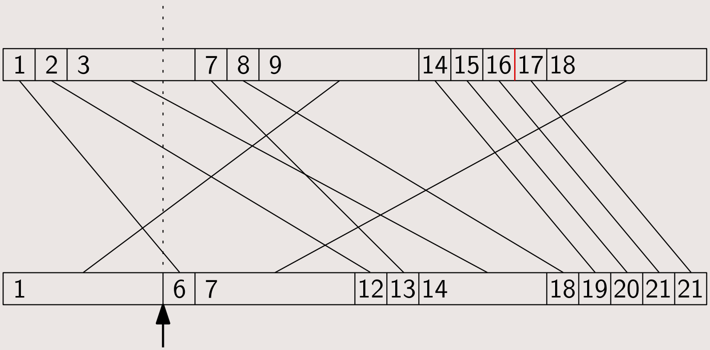


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals

Example Execution ($a = 2$)

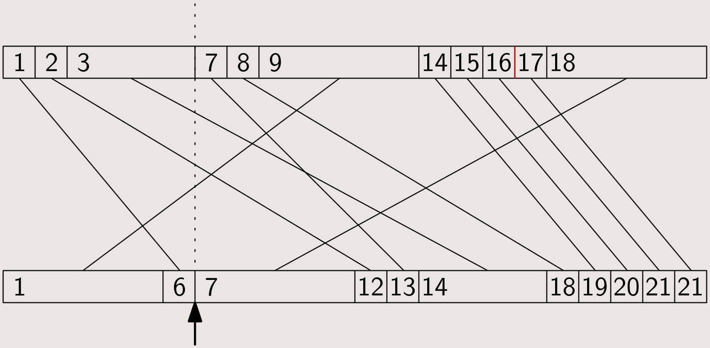


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals

Example Execution ($a = 2$)

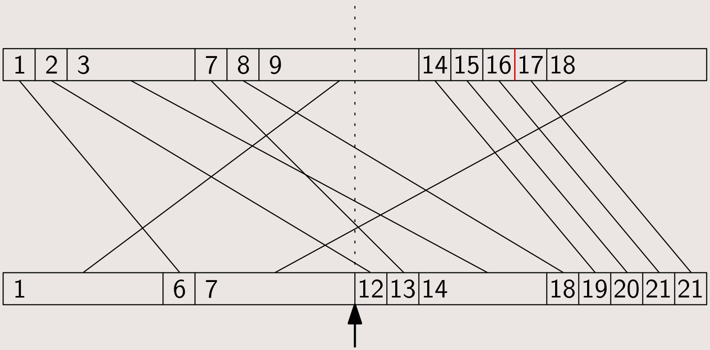


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals

Example Execution ($a = 2$)

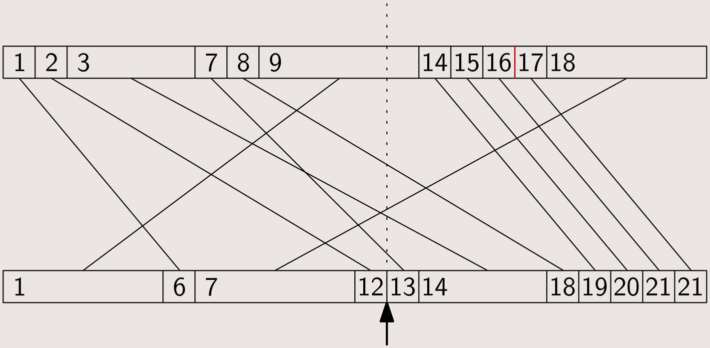


Fast Balancing Algorithm

General Approach

- Simultaneously iterate over input- and output intervals

Example Execution ($a = 2$)

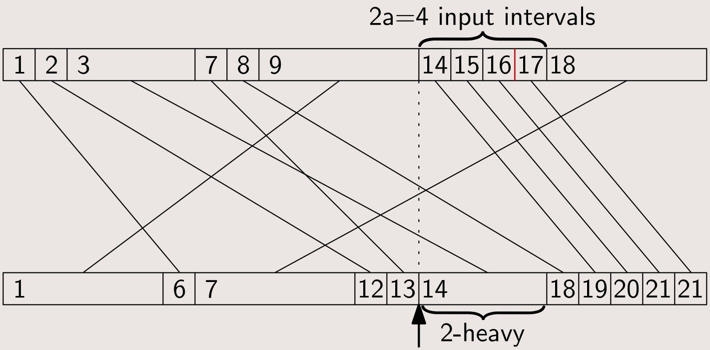


Fast Balancing Algorithm

General Approach

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

Example Execution ($a = 2$)

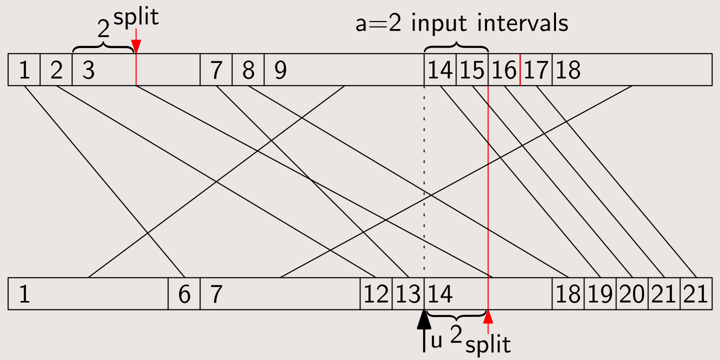


Fast Balancing Algorithm

General Approach

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

Example Execution ($a = 2$)

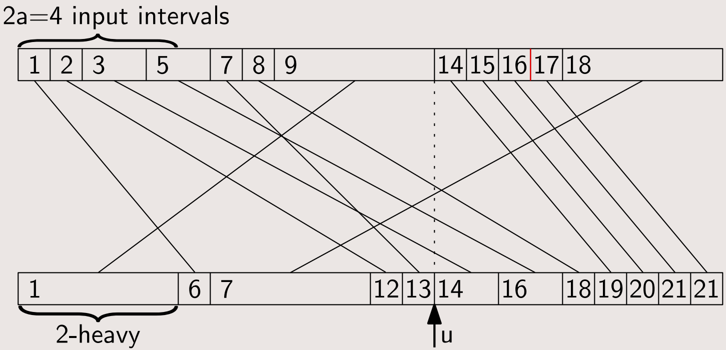


Fast Balancing Algorithm

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 ($a = 2$)

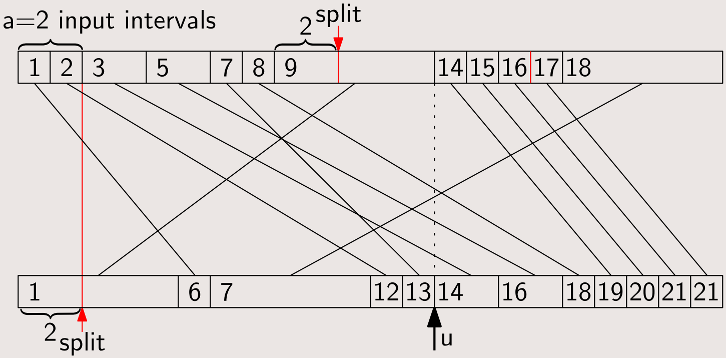


Fast Balancing Algorithm

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 ($a = 2$)

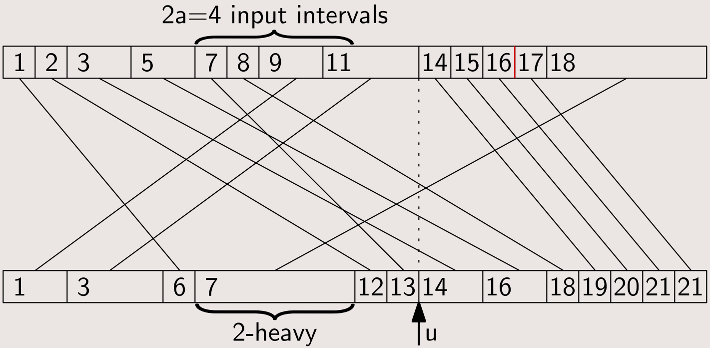


Fast Balancing Algorithm

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 ($a = 2$)

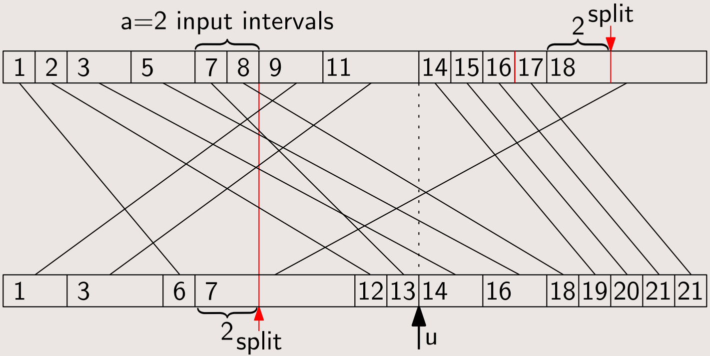


Fast Balancing Algorithm

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 ($a = 2$)

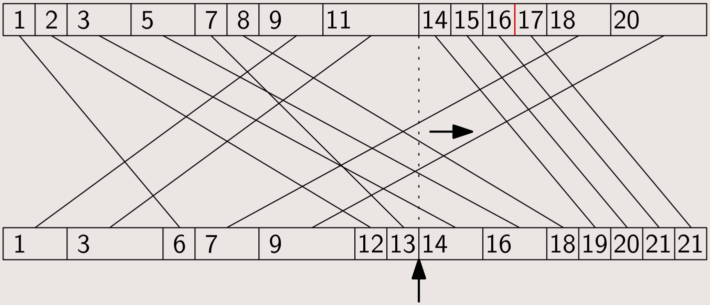


Fast Balancing Algorithm

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 ($a = 2$)

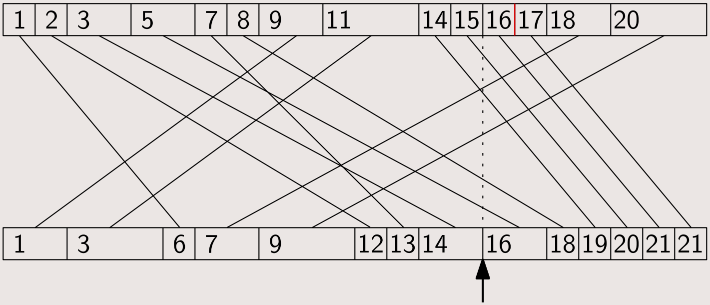


Fast Balancing Algorithm

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 ($a = 2$)

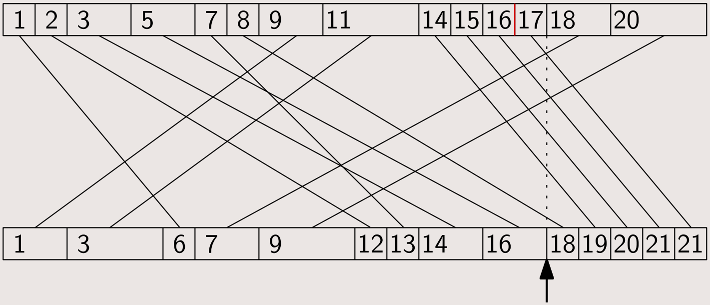


Fast Balancing Algorithm

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 ($a = 2$)

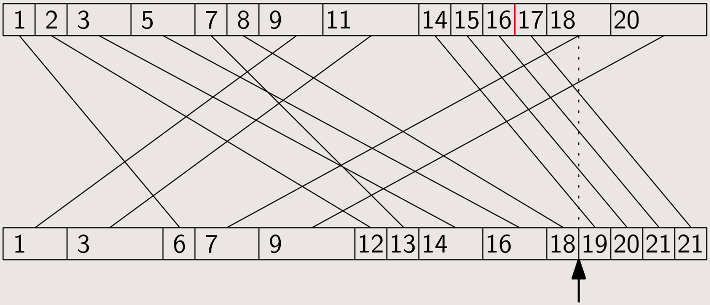


Fast Balancing Algorithm

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 ($a = 2$)

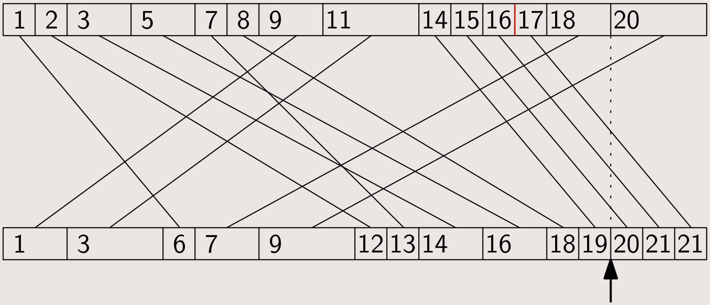


Fast Balancing Algorithm

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 ($a = 2$)

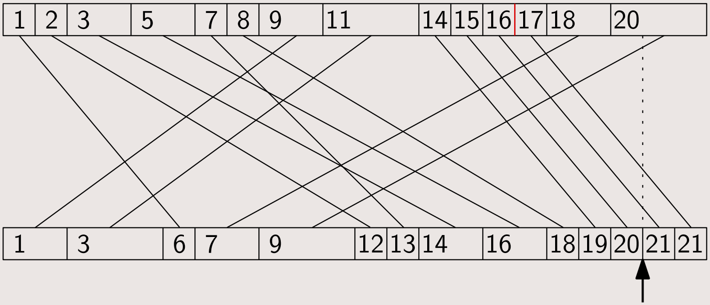


Fast Balancing Algorithm

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 ($a = 2$)

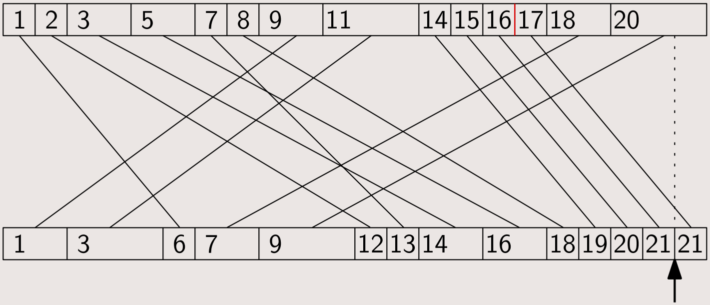


Fast Balancing Algorithm

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 ($a = 2$)

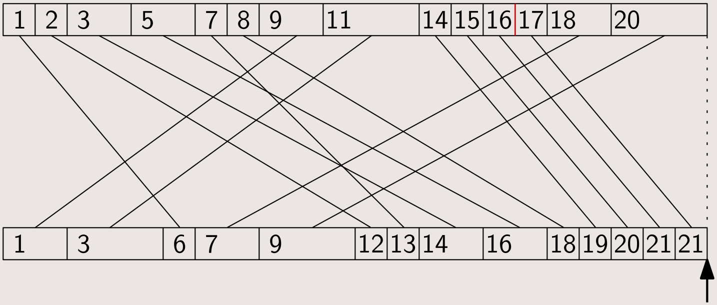


Fast Balancing Algorithm

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

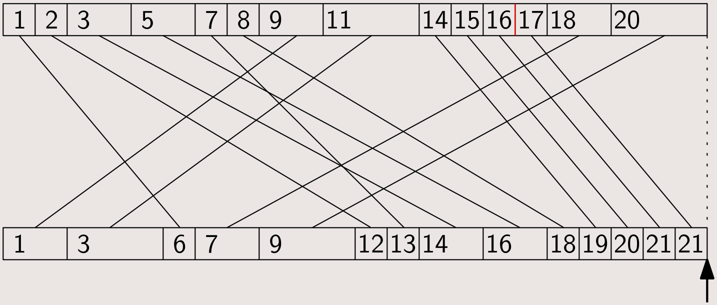
- Simultaneously iterate over input- and output intervals
- If an output interval is a-heavy:
  - Split and remember starting position u
  - Check for new a-heavy output interval before u and recurse

Details

- Use balanced search trees for input and output intervals

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

Example Execution ($a = 2$)



Experimental Setup

Tested Indexes

- move-r
- r-index [Gagie et al. 2020]
- online-rlbwt (dynamic) [Bannai et al. 2020]
- rcomp-glfig (dynamic) [Nishimoto et al. 2022]

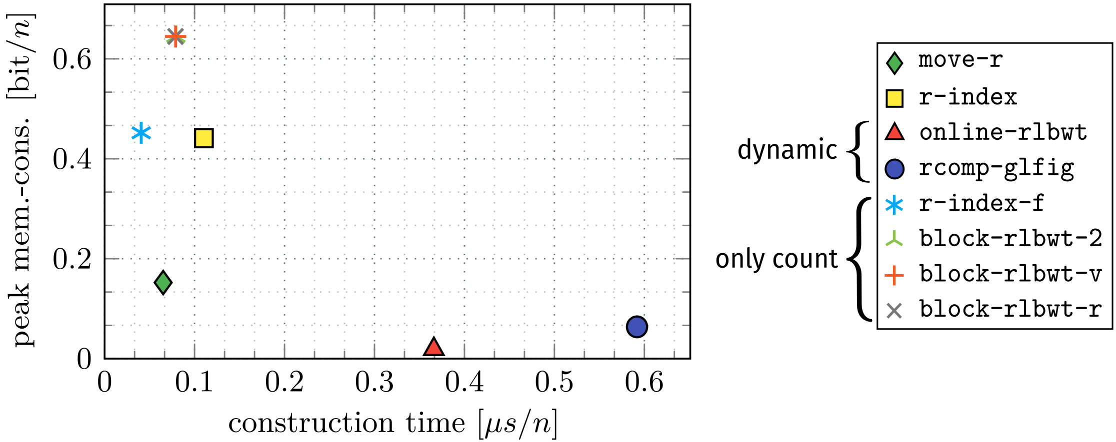
Experimental Setup

Tested Indexes

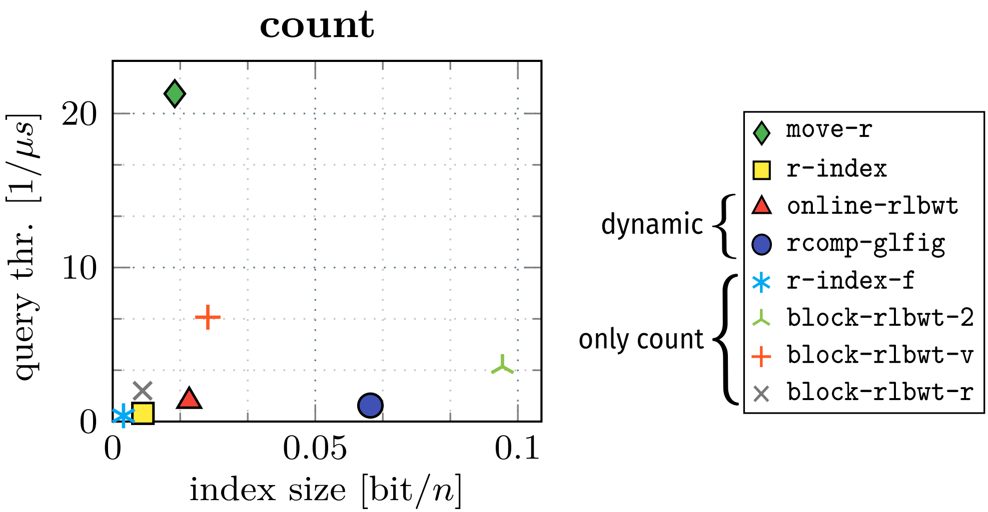
- ◆ move-r
- ■ r-index [Gagie et al. 2020]
- ▲ online-rlbwt (dynamic) [Bannai et al. 2020]
- ● rcomp-glfig (dynamic) [Nishimoto et al. 2022]
- ∗ r-index-f [Brown et al. 2022]
- ⅄ block-rlbwt-2 [Díaz-Domínguez et al. 2023]
- + block-rlbwt-v [Díaz-Domínguez et al. 2023]
- × block-rlbwt-r [Díaz-Domínguez et al. 2023]

only count

Construction Performance (einstein.en.txt)



Query Performance (einstein.en.txt)



Conclusion

Conclusion

- The Move Data Structure can be constructed efficiently in practice
- The improved $\mathcal{O}(1)$ time to answer LF- and Φ queries is reflected in practice

Conclusion

Conclusion

- The Move Data Structure can be constructed efficiently in practice
- The improved $\mathcal{O}(1)$ time to answer LF- and Φ queries is reflected in practice
- Optimizing other aspects of the r-index further improved construction- and query performance
  - 2x-35x (typ. 15x) faster queries
  - 0.9-2x (typ. 2x) faster construction with 1-3 (typ. 3x) lower memory usage, but
  - 0.8x-2.5x (typ. 2x) larger index