Move-r: Optimizing the r-index

Nico Bertram
Technische Universität Dortmund, Germany

Johannes Fischer
Technische Universität Dortmund, Germany

Lukas Nalbach
Technische Universität Dortmund, Germany

Abstract

We present a static text index called Move-r, which is a highly optimized version of the r-index ([11] Gagie et al., 2020) that encorporates recent theoretical developments of the *move data structure* ([19] Nishimoto and Tabei, 2021). The r-index is the method of choice for indexing highly repetitive texts, such as different versions of a text document or DNA from the same species, as it exploits the compressibilty of the underlying data. With Move-r, we can answer count- and locate queries 2–35 (typically 15) times as fast as with any other r-index supporting locate queries while being 0.8–2.5 (typically 2) times as large. A Move-r index can be constructed 0.9–2 (typically 2) times as fast while using 1/3–1 (typically 1/2) times as much space.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Compressed Text Index, Burrows-Wheeler Transform

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.1

Supplementary Material *Software (Source Code)*: `https://github.com/LukasNalbach/Move-r`
archived at `swh:1:dir:5dee6ddf67482e0fb6f4a2c4c7e9d743332a38a6`

1 Introduction

Answering pattern matching queries on repetitive texts is a common task in bioinformatics, in particular when indexing DNA (assembled or unassembled) from the same species. In such situations, it is important to exploit the repetitiveness of the data, and not use indexes that store the whole data uncompressed.

The r-index [11] is a recent and important development in this area, and uses $\mathcal{O}(r)$ words of space, where r is the number of equal-letter runs in the Burrows-Wheeler Transformation, which is an accepted measure for compressibility of highly similar texts [17]. The r-index answers counting queries (counting the number of matches of a length-m pattern) in $\mathcal{O}(m \log\log_w(\sigma + n/r))$ time, and locate queries (listing all matching positions) in additional $\mathcal{O}(\mathrm{occ} \log\log_\omega(n/r))$ time, where n is the total text length, occ is the number of matches, σ is the alphabet size, and w the word width of the word-RAM.

The bottleneck during the computation of those queries are predecessor queries. Recently, the so-called *move data structure* has been proposed [19], which can answer the r-index-specific predecessor queries in $\mathcal{O}(1)$ time. The move data structure uses the property that two important functions (LF and Φ, see next section for their definitions) can be divided into r intervals where the function values increase by exactly 1. Due to specific access patterns of those functions during a pattern query, it is possible to subdivide those intervals (a process called *balancing*) such that the functions can be evaluated in $\mathcal{O}(1)$ time, while still using only $\mathcal{O}(r)$ space.

Our Contributions. The practical performance of the move data structure is still largely open: how can it be constructed efficiently, how fast does it answer queries, and how does it compare with other implementations of the r-index? Does it even lead to a practical advantage? We answer these questions and make the following contributions with a data structure that we call Move-r:

- We present a fast construction algorithm for the move data structure (Section 4.1). While the general idea of the move data structure is simple to implement, it is not obvious how to practically perform the balancing step (Section 3.2) efficiently.
- We present a practically optimized locate-algorithm (Section 4.2) that significantly reduces the number of memory accesses, at the cost of an asymptotically worse running time.
- We introduce additional practical improvements to the move data structure that reduce the space and improve cache efficiency (Appendix B).
- From our experiments we conclude that the theoretical benefit of the $\mathcal{O}(1)$ time over the $\mathcal{O}(\log\log_\omega(n/r))$ time to answer predecessor queries is reflected in practice, since although Move-r indexes are 0.8–2.5 (typically 2) times as large as the fastest other r-index supporting locate queries, they can answer count- and locate queries 2–35 (typically 15) times as fast. Move-r can also be constructed 0.9–2 (typically 2) times as fast (and 2.5–30, typically 20, times as fast as any *dynamic* r-index), while leaving a memory footprint that is 1/3–1 (typically 1/2) times as large as those of other static r-indexes and sometimes even competitive with dynamic r-indexes.

2 Preliminaries

In the following, we define necessary notations for the rest of this paper. An *interval* $[i, j]$ describes the set $\{i, ..., j\}$. For convenience, we use the following notations: $[i, j] = [i, j+1) = (i-1, j] = (i-1, j+1)$. Given a set $S \subset U$, we define the *predecessor* of $i \in U$ in S as $\text{pred}(i) = \max\{j | j \in S \wedge j < i\}$. Similarly, we define the *successor* of $i \in U$ in S as $\text{succ}(i) = \min\{j | j \in S \wedge j > i\}$. We assume that all described algorithms work in the *word-RAM model* with word size w [13].

An *alphabet* Σ is a finite ordered set of *symbols* of size $|\Sigma| = \sigma$. A string $T \in \Sigma^*$ with $|T| = n$ is a sequence of symbols in the alphabet Σ. In case that $|T| = 0$, T is the empty string ϵ. We can access the i-th symbol of T with $T[i]$. A *substring* $T[i, j]$ is defined by $T[i] \cdot T[i+1] \cdot ... \cdot T[j]$. In case that $i > j$, we set $T[i, j] = \epsilon$. The substring $T[1, i]$ is called the i-th *prefix* of T and the substring $T[i, n]$ is called the i-th *suffix* of T and is denoted by T_i. In case that the length of a prefix or suffix is at least 1 and at most $n - 1$ we call them *proper*. We assume an order $c_1 < c_2 < ... < c_\sigma$ on the alphabet $\Sigma = \{c_1, c_2, ..., c_\sigma\}$. The *lexicographic order* of strings is then defined by $T < T' \Leftrightarrow T$ is a proper prefix of $T' \vee \exists i : T[1, i] = T'[1, i] \wedge T[i+1] < T'[i+1]$. We are also interested in how often a character c appears in a string $T[1, i]$, which we denote by $\mathsf{rank}(T, c, i)$.

2.1 Burrows-Wheeler Transform

In the following, each string $T \in \Sigma^*$ is terminated by a *sentinel* symbol \$ that is lexicographically smaller than all other symbols in Σ. This allows us to simplify our algorithms.

The *suffix array* [14] SA of a string T consists of the starting positions of all suffixes of T in their lexicographic order, i.e, $T_{\mathsf{SA}[1]} < T_{\mathsf{SA}[2]} < ... < T_{\mathsf{SA}[n]}$. For a pattern $P \in \Sigma^*$, there is a maximum interval in the suffix array (the *suffix array interval* of P) containing exactly the positions of all occurrences of P in T. For a string T, the i-th *rotation* is defined by $T[i, n]T[1, i-1]$. The *rotation matrix* is then the $n \times n$ matrix where the i-th row consists of the i-th rotation of T. When we sort the rotation matrix lexicographically, the last column L is then called the *Burrows-Wheeler Transform* (BWT) [6] of T and the first column is denoted by F. The BWT has the property that same symbols are often grouped *runs* of the same symbols. For that reason, we can compress the BWT by applying a *run-length encoding* to it. Let $L_1, L_2, ..., L_r$ denote those runs, i.e, r is the number of runs in L and $L = L_1 \cdot L_2 \cdot ... \cdot L_r$. Then we can compress the BWT into the *run-length encoded BWT* (RLBWT) $(L_1[1], |L_1|), (L_2[1], |L_2|), ..., (L_r[1], |L_r|)$.

The *last-to-first* (LF) mapping of a string maps each symbol in the BWT to the corresponding symbol in F and is defined by $\mathsf{LF}(i) = j \Leftrightarrow \mathsf{SA}[j] = \mathsf{SA}[i] - 1$ if $\mathsf{SA}[i] > 1$, and $\mathsf{LF}(i) = 1$ if $\mathsf{SA}[i] = 1$. The LF *property* describes that the order of same symbols is identical in F and L, i.e, $L[i] = L[j] \wedge i < j \Rightarrow \mathsf{LF}(i) < \mathsf{LF}(j)$. Given the BWT L of a string T, we can augment L with a rank data structure and the frequency array C of each character with $C[c] = |\{i \in [1, n] \mid T[i] < c\}|$. Then, $\mathsf{LF}(i) = C[L[i]] + \mathrm{rank}(L, L[i], i)$ because equal symbols in F are grouped together, and the LF property. Thus, the following important property holds, which allows us to implement LF in $\mathcal{O}(r)$ space:

▶ **Lemma 1.** *Let T be a string of length n and L be its BWT. Let $l_1 < l_2 < ... < l_r$ be the starting positions of each run in L and let $l_{r+1} = n + 1$. Then $\mathsf{LF}(i) = \mathsf{LF}(l_x) + (i - l_x)$ holds for $l_x \leq i < l_{x+1}$.*

2.2 Backward Search

We can answer count queries for a pattern P of length m by using a *backward search* [10]. We search for P by iterating from right to left over P and determining in iteration i (implicitly) the suffix array interval $[b, e]$ that is prefixed by $P[m-i+1, m]$. We initialize $[b, e]$ with $[1, n]$. Now, consider the i-th iteration. Let $c = P[m - i + 1]$ be the character under consideration. We can find the suffix array interval that is prefixed by c by accessing $C[c]$. By using the LF-property, we have to jump over all occurrences of c in L before $[b, e]$ by using rank queries. So, we set $b = C[c] + \mathrm{rank}(L, c, b - 1) + 1$ and $e = C[c] + \mathrm{rank}(L, c, e)$. Having read P completely, the output $|[b, e]|$ is the number of occurrences of P in T. If $|[b, e]| = 0$, there are no occurrences of P in T. If the suffix array is at hand, we can also answer locate queries by outputting all starting positions of the occurrences in T.

A main insight of the r-index is the following: If we know the last value of a suffix array interval, we can reconstruct the values in the entire interval without explicitly storing SA. This is formalized by the function Φ, where $\Phi(\mathsf{SA}[i]) = \mathsf{SA}[i - 1]$ for $i > 1$ and $\Phi(\mathsf{SA}[1]) = \mathsf{SA}[n]$. The following property holds, which allows us to implement Φ in $\mathcal{O}(r)$ space – note the similarity to Lemma 1.

▶ **Lemma 2** ([11], lemma 3.5)**.** *Let $\{u_1, u_2, ..., u_{r+1}\} = \{\mathsf{SA}[l_1], \mathsf{SA}[l_2], ..., \mathsf{SA}[l_r], n + 1\}$ and $u_1 < u_2 < ... < u_{r+1} = n + 1$. Then $\Phi(i) = \Phi(u_x) + (i - u_x)$ for $u_x \leq i < u_{x+1}$.*

3 The Move Data Structure

We now describe the *move data structure* [19]. By Lemma 1, the LF function can be divided into r blocks with starting positions $l_1, l_2, ..., l_r$ and $l_{r+1} = n + 1$. Similarly, we can divide Φ into r blocks (see Lemma 2) with starting positions $u_1, u_2, ..., u_r$. Thus, we only need to store the function values for each block head, and can compute every other value by adding the offset between the value and the block head. In the following we will generalize this concept to *interval sequences* [19] and describe how to efficiently answer queries on interval sequences.

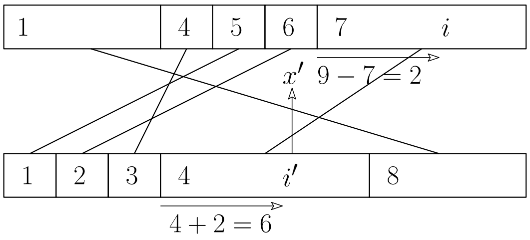

Figure 1 Visualization of the disjoint interval sequence $I = (1,8),(4,3),(5,1),(6,2),(7,4)$ with $n = 10$, and the evaluation of the query $\mathsf{move}(9,5) = (6,4)$.

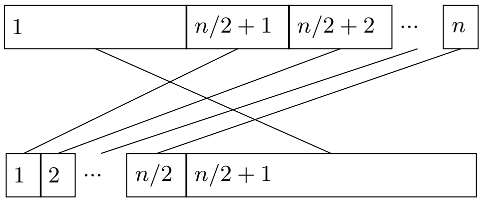

Figure 2 When evaluating the query $\mathsf{move}(n/2,1) = (n,k)$ on this sequence, we set $x' \leftarrow \mathcal{M}_{\mathsf{idx}}[1] = 2$ and increment x' $k-2$ times until we get $x' = k$.

3.1 Disjoint Interval Sequence

We first formally define *disjoint interval sequences.*

▶ **Definition 3.** *Let $I = (p_1,q_1),(p_2,q_2),...,(p_k,q_k)$ be a sequence of pairs of elements in the range $[1,n]$, where $q_1,q_2,...,q_k$ are pairwise distinct, let π be the permutation of $[1,k]$ s.t. $1 = q_{\pi[1]} < q_{\pi[2]} < ... < q_{\pi[k]} \le n$, let $p_{k+1} = n+1 = q_{k+1}$, and let $d_i = p_{i+1} - p_i$ for $i \in [1,k]$. Then, I is called a disjoint interval sequence if $1 = p_1 < p_2 < ... < p_k < p_{k+1}$ and $d_{\pi[i]} = q_{\pi[i+1]} - q_{\pi[i]}$ for all $i \in [1,k]$.*

A disjoint interval sequence defines a sequence of *input intervals* and *output intervals.* The input intervals consist of the ranges $[p_i, p_i + d_i)$, and the output intervals consist of the ranges $[q_i, q_i + d_i)$, where the input interval starting with p_i and the output interval starting with q_i have the same size d_i. It describes the bijective function f_I that maps a position in an input interval to the position with the same offset in its corresponding output interval by $f_I(i) = q_x + (i - p_x)$, where x is the index of the input interval containing the position i. See Figure 1 for an example. The intervals at the top represent the input intervals and the intervals at the bottom represent the output intervals. A line between an input interval and an output interval is drawn between corresponding intervals and represents the permutation π. A *move query* $\mathsf{move}(i,x) = (i',x')$ evaluates for a given value $i \in [1,n]$ and the index $x \in [1,k]$ of the input interval containing i the function $f_I(i) = i'$, and also returns the index x' of the input interval containing i'. x' can then be used to directly evaluate the next query $\mathsf{move}(i',x')$. See again Figure 1 for an example.

To answer move queries on I, we use a move data structure $\mathcal{M}$, which stores three arrays $\mathcal{M}_{\mathsf{p}}[1..k] = [p_1,p_2,...,p_k]$, $\mathcal{M}_{\mathsf{q}}[1..k] = [q_1,q_2,...,q_k]$ and $\mathcal{M}_{\mathsf{idx}}[1..k]$, where $\mathcal{M}_{\mathsf{idx}}$ stores at position j the index of the input interval containing q_j, i.e, $\mathcal{M}_{\mathsf{idx}}[j] = i$ iff $q_j \in [p_i, p_i + d_i)$. This allows us to compute the i'-part of the query $\mathsf{move}(\mathcal{M},i,x) = (i',x')$ by $i' = \mathcal{M}_q[x] + (i - \mathcal{M}_p[x])$. To find x', we set $x' \leftarrow \mathcal{M}_{\mathsf{idx}}[x]$. Now, we have to iterate over the starting positions of the input intervals while incrementing x' to find the input interval containing i'. In the worst case this can take $\mathcal{O}(k)$ steps (see Figure 2). In the following, we show how to reduce the time to answer a move query to $\mathcal{O}(a)$ for a fixed parameter a.

3.2 Balancing a disjoint interval sequence

Now, we show how to balance a disjoint interval sequence, which guarantees constant time to answer move queries.

▶ **Definition 4.** *We call an output interval* $[q_j, q_j + d_j)$ *of* I a-heavy *if there are at least* $2a$ *input intervals of* I *starting in it. Otherwise, we call it* a-balanced*. We call a disjoint interval sequence* a-balanced *iff all of its output intervals are* a*-balanced. Otherwise, we call it* a-heavy*.*

The general idea of a balancing algorithm is to find an a-heavy output interval and split it (and its corresponding input interval) into two new intervals, each. The first new output interval will be a-balanced, and the second new output interval is either a-heavy or a-balanced. By iteratively applying this procedure (the a-balancing step), we obtain a disjoint interval sequence that is a-balanced.

▶ **Definition 5.** *We define an* a-balancing step *on* I *to be the process of splitting an* a*-heavy output interval* $[q_j, q_j + d_j)$ *of* I *and its corresponding input interval* $[p_j, p_j + d_j)$ *at the offset* $d = p_{i+a} - q_j$ *to obtain the disjoint interval sequence* $I' = (p_1, q_1), ..., (p_j, q_j), (p_j + d, q_j + d), (p_{j+1}, q_{j+1}), ..., (p_k, q_k)$*, where* $[p_i, p_i + d_i)$ *is the first input interval of* I *starting in* $[q_j, q_j + d_j)$*. We call an algorithm* $\mathcal{A}$ *a* balancing algorithm*, if* $\mathcal{A}$ *takes a disjoint Interval sequence* I *and an integer* $a \geq 2$ *as input, iteratively performs* a*-balancing steps starting with* I *until the disjoint interval sequence is* a*-balanced, and finally returns the resulting disjoint interval sequence.*

Now we show that every balancing algorithm terminates and that the resulting disjoint interval sequence is at most $\frac{a}{a-1}$ times as large as I.

▶ **Theorem 6** (Generalization of Lemma 6 [19])**.** *Let* I *be a disjoint interval sequence of size* k*, let* $a \geq 2$ *be an integer, let* $\mathcal{A}$ *be a balancing algorithm and let* $\mathcal{B}_a(I)$ *be the disjoint interval sequence resulting from an execution of* $\mathcal{A}$ *on* I*. Then* $f_{\mathcal{B}_a(I)} = f_I$ *and* $|\mathcal{B}_a(I)| \leq \frac{a}{a-1}k$*.*

Proof. Let t' be the number of a-balancing steps $\mathcal{A}$ performs during this execution, let $t \in [1, t']$, let I_t be the disjoint interval sequence after the t-th step, let $I_0 = I$, let $[q_j, q_j + d_j)$ be the output interval of I_{t-1} that $\mathcal{A}$ splits in the t-th step, and let $[p_i, p_i + d_i)$ be the first input interval of I_{t-1} starting in $[q_j, q_j + d_j)$.

Since $\mathcal{A}$ splits $[p_j, p_j + d_j)$ and $[q_j, q_j + d_j)$ at the same offset $d = p_{i+a} - q_j$ to obtain I_t from I_{t-1}, clearly $|I_t| = |I_{t-1}| + 1$ and $f_{I_t} = f_{I_{t-1}}$ and therefore $|I_t| = k + t$ and $f_I = f_{I_0} = f_{I_1} = ... = f_{I_{t'}} = f_{\mathcal{B}_a(I)}$.

Let $O_{a,t}$ be the set of output intervals in I_t containing at least a input intervals of I_t. We have $O_{a,t} \supseteq O_{a,t-1} \setminus \{[q_j, q_j + d_j)\} \cup \{[q_j, q_j + d), [q_j + d, q_j + d_j)\}$, which implies $|O_{a,t}| \geq |O_{a,t-1}| + 1$ and therefore $|O_{a,t}| \geq t$. Since the output intervals in $O_{a,t}$ are pairwise disjoint, there cannot be less than $a|O_{a,t}|$ input intervals in I_t, which yields $k + t = |I_t| \geq a|O_{a,t}| \geq at \Leftrightarrow t \leq \frac{k}{a-1}$ and therefore $|\mathcal{B}_a(I)| = k + t' \leq \frac{a}{a-1}k$. ◀

Recall that the dominating part of answering a query $\mathsf{move}(i, x) = (i', x')$ is to determine x' by starting at the first input interval starting in the output interval $[q_x, q_x + d_x)$ and iterating over possibly all other input intervals starting in $[q_x, q_x + d_x)$. Since in $\mathcal{B}_a(I)$, the number of these input intervals is bounded by $2a - 1$, the following corollary holds.

▶ **Corollary 7** (The Move Data Structure)**.** *Let* I *be a disjoint interval sequence of length* k *and* $a \geq 2$ *be an integer. Then there is a data structure of size* $\mathcal{O}(\frac{a}{a-1}k)$ *that allows us to consecutively evaluate* f_I *in* $\mathcal{O}(a)$ *time using move queries.*

4 Algorithmic Optimizations

At first, we present our new efficient algorithm to balance disjoint interval sequences. Then, we discuss Move-r and compare it with Nishimoto and Tabei's *OptBWTR* [19] index. In Appendix B, we describe discuss further practical improvements.

4.1 New algorithm to balance disjoint interval sequences

For constructing an a-balanced disjoint interval sequence given a disjoint interval sequence, we use two balanced search-trees $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ storing the pairs of I sorted by the starting positions of the input intervals and the output intervals, respectively. We add $(n+1, n+1)$ to both s.t. for every pair in $\mathcal{T}_{\mathsf{in}}$ or $\mathcal{T}_{\mathsf{out}}$ we can compute the length of the input- or output-interval it creates by accessing the next pair in $\mathcal{T}_{\mathsf{in}}$ or $\mathcal{T}_{\mathsf{out}}$, respectively.

To speed up the construction of $\mathcal{T}_{\mathsf{out}}$, we at first calculate the permutation $\pi[1..k]$ from the definition of the disjoint interval sequence by sorting any permutation of $[1, k]$ by the starting positions of the output intervals of I. Since $\pi[i]$ points to the pair creating the i-th output interval, we can build $\mathcal{T}_{\mathsf{out}}$ in $\mathcal{O}(k)$ additional time by inserting the pairs in the order $(p_{\pi[1]}, q_{\pi[1]}), (p_{\pi[2]}, q_{\pi[2]}), ..., (p_{\pi[k]}, q_{\pi[k]})$ into $\mathcal{T}_{\mathsf{out}}$.

Let $\mathcal{T}$ be a balanced search-tree over a set X. Then $\langle v \rangle$ denotes the node in $\mathcal{T}$ storing $v \in X$, $\mathcal{T}.\mathsf{has\text{-}next}(\langle v \rangle)$ returns whether $\langle v \rangle$ has a successor node in $\mathcal{T}$, $\mathcal{T}.\mathsf{next}(\langle v \rangle)$ returns this successor node, $\mathcal{T}.\mathsf{next}(\langle v \rangle, n)$ returns the n-th successor of $\langle v \rangle$ in $\mathcal{T}$, $\mathcal{T}.\mathsf{pred}(v) = \langle \max\{w \in \mathcal{T} \mid w \leq v\} \rangle$, $\mathcal{T}.\mathsf{succ}(v) = \langle \min\{w \in \mathcal{T} \mid v \leq w\} \rangle$, $\mathcal{T}.\mathsf{min}() = \langle \min\{v \in \mathcal{T}\} \rangle$ and $\langle p, q \rangle_{\mathsf{in}}$, and $\langle p, q \rangle_{\mathsf{out}}$ denote the nodes in $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ storing (p, q), respectively. The algorithm $\mathsf{is\text{-}a\text{-}heavy}$ expects an output interval $[q_j, q_j + d_j)$ and $\langle p_i, q_i \rangle_{\mathsf{in}}$, where $[p_i, p_i + d_i)$ is the first input interval starting in $[q_j, q_j + d_j)$. It iterates forward at most $2a$ steps in $\mathcal{T}_{\mathsf{in}}$ starting with $\langle p_i, q_i \rangle_{\mathsf{in}}$ and returns whether $[q_j, q_j + d_j)$ is a-heavy.

4.1.1 Framework Balancing Algorithm

The general idea of our balancing algorithm is to not explicitly store all a-heavy output intervals in a data structure, but to compute and balance them on-the-fly. We achieve this by implementing a deterministic balancing algorithm that in each step chooses to balance the output interval with the smallest starting position. Algorithm 1 is the framework of our balancing algorithm.

In the iterations of main **while**-loop (lines 3-13), we consider the pairs $(p_i, q_i) \in \mathcal{T}_{\mathsf{in}}$ and $(p_j, q_j) \in \mathcal{T}_{\mathsf{out}}$ ascendingly ordered by q_j, where i is minimal s.t. $q_j \leq p_i < q_j + d_j$ holds, if such i exists, i.e, we consider the output intervals $[q_j, q_j + d_j)$ in ascending order of their starting position together with the respective first input intervals $[p_i, p_i + d_i)$ starting in them. We initialize $\langle p_i, q_i \rangle_{\mathsf{in}}$ and $\langle p_j, q_j \rangle_{\mathsf{out}}$ with the first pairs in $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$, respectively. For those, $q_j \leq p_i < q_j + d_j$ trivially holds, since $q_{\pi[1]} = p_1 = 1 < q_{\pi[1]} + d_{\pi[1]}$.

Given some combination of such pairs $\langle p_i, q_i \rangle_{\mathsf{in}}$ and $\langle p_j, q_j \rangle_{\mathsf{out}}$, we want to find the next combination of such pairs, that is we want to find $(p_{i'}, q_{i'}) \in \mathcal{T}_{\mathsf{in}}$ and $(p_{j'}, q_{j'}) \in \mathcal{T}_{\mathsf{out}}$, where $q_{j'}$ is minimal s.t. $q_{j'} > q_j$ and there exists a pair $(p_{i'}, q_{i'}) \in \mathcal{T}_{\mathsf{in}}$, where $p_{i'}$ is minimal s.t. $q_{j'} \leq p_{i'} < q_{j'} + d_{j'}$. To find those, we at first iterate further one step with $\langle p_j, q_j \rangle_{\mathsf{out}}$ in $\mathcal{T}_{\mathsf{out}}$ (line 8). We then iterate (possibly zero steps) with $\langle p_i, q_i \rangle_{\mathsf{in}}$ in $\mathcal{T}_{\mathsf{in}}$, until $p_i > q_j$ holds, or we have reached the second to last pair in $\mathcal{T}_{\mathsf{in}}$ (lines 10-12). If then also $p_i < q_j + d_j$ holds (line 13), we have found the next combination of such pairs. Else, we again alternately iterate further as described with $\langle p_i, q_i \rangle_{\mathsf{in}}$ and $\langle p_j, q_j \rangle_{\mathsf{out}}$ until we find such pairs or until we reach the second to last pair in $\mathcal{T}_{\mathsf{in}}$ or $\mathcal{T}_{\mathsf{out}}$, respectively.

Algorithm 1 balance-framework().

1 $\langle p_i, q_i\rangle_{\mathsf{in}} \leftarrow \mathcal{T}_{\mathsf{in}}$.min();
2 $\langle p_j, q_j\rangle_{\mathsf{out}} \leftarrow \mathcal{T}_{\mathsf{out}}$.min();
3 **while true do**
4   **if** is-a-heavy($[q_j, q_j + d_j), \langle p_i, q_i\rangle_{\mathsf{in}}$) **then**
5   balance-up-to($\langle p_i, q_i\rangle_{\mathsf{in}}, \langle p_j, q_j\rangle_{\mathsf{out}}, q_j$);
6   **else**
7   **do**
8             $\langle p_j, q_j\rangle_{\mathsf{out}} \leftarrow \mathcal{T}_{\mathsf{out}}$.next($\langle p_j, q_j\rangle_{\mathsf{out}}$);
9   **if** $\neg\mathcal{T}_{\mathsf{out}}$.has-next($\langle p_j, q_j\rangle_{\mathsf{out}}$) **then return**;
10   **while** $p_i < q_j$ **do**
11                 $\langle p_i, q_i\rangle_{\mathsf{in}} \leftarrow \mathcal{T}_{\mathsf{in}}$.next($\langle p_i, q_i\rangle_{\mathsf{in}}$);
12   **if** $\neg\mathcal{T}_{\mathsf{in}}$.has-next($\langle p_i, q_i\rangle_{\mathsf{in}}$) **then return**;
13   **while** $q_j + d_j \leq p_i$;

We want to make sure that each output interval $[q_j, q_j + d_j)$ considered in the main **while**-loop is a-balanced. To do this, we check if $[q_j, q_j + d_j)$ is a-heavy by calling is-a-heavy($[q_j, q_j + d_j), \langle p_i, q_i\rangle_{\mathsf{in}}$) in line 4. If it is a-balanced, we proceed to iterate further over $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ as described to find the next combination of $\langle p_i, q_i\rangle_{\mathsf{in}}$ and $\langle p_j, q_j\rangle_{\mathsf{out}}$ (lines 7-13). Else, we call balance-up-to($\langle p_i, q_i\rangle_{\mathsf{in}}, \langle p_j, q_j\rangle_{\mathsf{out}}, q_j$) in line 5, which performs at least one a-balancing step. More precisely, it splits $[q_j, q_j + d_j)$ at the offset $d = p_{i+a} - q_j$ into two new output intervals $[q_j, q_j + d)$ and $[q_j + d, q_j + d_j)$, then possibly recursively a-balances all output intervals starting before q_j becoming a-heavy in the process. We will prove the correctness of balance-up-to in the next section (see Theorem 9). After each iteration of the main **while**-loop, all output intervals starting before q_j are a-balanced, hence, after the main **while**-loop, all output intervals are a-balanced.

4.1.2 Balancing Step

In Algorithm 1, we always call balance-up-to with $q_j = q_u$ (the non-recursive case). Inside balance-up-to, we will (possibly) recursively call balance-up-to with $q_j < q_u$ only (the recursive case). At first, we prove the correctness of balance-up-to in the recursive case. Let t' denote the number of balancing steps our balancing algorithm performs.

Algorithm 2 balance-up-to($\langle p_i, q_i\rangle_{\mathsf{in}}, \langle p_j, q_j\rangle_{\mathsf{out}}, q_u$).

$\langle p_{i+a}, q_{i+a}\rangle_{\mathsf{in}} \leftarrow \mathcal{T}_{\mathsf{in}}$.next($\langle p_i, q_i\rangle_{\mathsf{in}}, a$);
$d \leftarrow p_{i+a} - q_j$;
$\mathcal{T}_{\mathsf{in}}$.insert($(p_j + d, q_j + d)$);
$\mathcal{T}_{\mathsf{out}}$.insert($(p_j + d, q_j + d)$);
if $p_j + d < q_u$ **then**
    $\langle p_y, q_y\rangle_{\mathsf{out}} \leftarrow \mathcal{T}_{\mathsf{out}}$.pred($p_j + d$);
    $\langle p_z, q_z\rangle_{\mathsf{in}} \leftarrow \mathcal{T}_{\mathsf{in}}$.succ($q_y$);
    if is-a-heavy($[q_y, q_y + d_y), \langle p_z, q_z\rangle_{\mathsf{in}}$) **then**
        balance-up-to($\langle p_z, q_z\rangle_{\mathsf{in}}, \langle p_y, q_y\rangle_{\mathsf{out}}, q_u$);

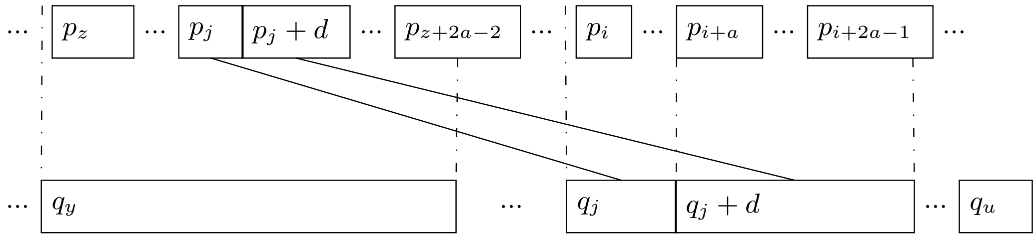

Figure 3 Illustration of an a-balancing step, where $p_j + d < q_u$ and $O_y = [q_y, q_y + d_y)$ is a-heavy.

▶ **Lemma 8.** *Let $s \in [0, t')$, let $[q_j, q_j + d_j)$ be an a-heavy output interval of I_s and let $q_u > q_j$, where $[q_j, q_j + d_j)$ contains exactly $2a$ input intervals of I_s, and all other output intervals of I_s starting before q_u are a-balanced. Let $[p_i, p_i + d_i)$ be the first input interval of I_s starting in $[q_j, q_j + d_j)$, let $t \in (s, t']$ be minimal s.t. all output intervals in I_t starting before q_u are a-balanced, and assume $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ store I_s. Then calling* balance-up-to($\langle p_i, q_i \rangle_{\mathsf{in}}, \langle p_j, q_j \rangle_{\mathsf{out}}, q_u$) *updates $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ to store I_t.*

Proof. We at first update $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ to store I_{s+1} by finding $\langle p_{i+a}, q_{i+a} \rangle_{\mathsf{in}}$ and inserting $(p_j + d, q_j + d)$ into $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$, where $d = p_{i+a} - q_j$. Compared with I_s, there are two new output intervals $[q_j, q_j + d)$ and $[q_j + d, q_j + d_j)$ in I_{s+1}. Both contain exactly a input intervals, hence they are a-balanced. Let $O_y = [q_y, q_y + d_y)$ be the output interval of I_{s+1} containing $p_j + d$. Since $p_j + d \in O_y$, O_y is the only possibly a-heavy output interval in I_{s+1} starting before q_u. If the **if**-clause in line 5 fails, then O_y starts at or after q_u, hence $I_t = I_{s+1}$ and we are done.

Else, let $[p_z, p_z + d_z)$ be the first input interval of I_{s+1} starting in O_y. We now find $\langle p_y, q_y \rangle_{\mathsf{out}}$ with a predecessor search over $\mathcal{T}_{\mathsf{out}}$ and $\langle p_z, q_z \rangle_{\mathsf{in}}$ using a successor search over $\mathcal{T}_{\mathsf{in}}$. Then, we check if O_y is a-heavy in I_{s+1} by calling is-a-heavy($[q_y, q_y + d_y), \langle p_z, q_z \rangle_{\mathsf{in}}$). If it is a-balanced, then $I_t = I_{s+1}$ and we are done.

If it is a-heavy, then there are exactly $2a$ input intervals of I_{s+1} starting in O_y, because O_y is a-balanced in I_s and compared with I_s, the number of input intervals of I_{s+1} starting in O_y has increased by exactly one (see Figure 3). Therefore, calling balance-up-to($\langle p_z, q_z \rangle_{\mathsf{in}}$, $\langle p_y, q_y \rangle_{\mathsf{out}}, q_u$) in line 9 satisfies the requirements of Lemma 8, hence we can iteratively apply the same overall argument to obtain I_{s+2} from I_{s+1}, I_{s+3} from I_{s+2}, etc. in the inner recursive calls of balance-up-to. Let balance-up-to($\langle p_{i'}, q_{i'} \rangle_{\mathsf{in}}, \langle p_{j'}, q_{j'} \rangle_{\mathsf{out}}, q_u$) be the $t - s - 1$-th such recursive call and let $O_{y'} = [q_{y'}, q_{y'} + d_{y'})$ be the output interval of I_t containing $p_{j'} + d'$, where $d' = p_{i'+a} - q_{j'}$. In this call, we update $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ to store I_t. By the definition of I_t, $O_{y'}$ either starts at or after q_u or is a-balanced in I_t, hence one of the **if**-clauses fails and the series of recursive calls ends. ◀

With Lemma 8, we can now prove the correctness of balance-up-to in the non-recursive case ($q_j = q_u$).

▶ **Theorem 9.** *Let $s \in [0, t')$, let $[q_j, q_j + d_j)$ be the first a-heavy output interval of I_s and let $[p_i, p_i + d_i)$ be the first input interval of I_s starting in $[q_j, q_j + d_j)$. Let $t \in (s, t']$ be minimal s.t. all output intervals in I_t starting before q_j are a-balanced, and assume $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ store I_s. Then calling* balance-up-to($\langle p_i, q_i \rangle_{\mathsf{in}}, \langle p_j, q_j \rangle_{\mathsf{out}}, q_j$) *updates $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ to store I_t.*

Proof. As in the proof of Lemma 8, let $O_y = [q_y, q_y + d_y)$ be the output interval of I_{s+1} containing $p_j + d$. If O_y starts at or after q_j or if O_y is a-balanced in I_{s+1}, then $I_t = I_{s+1}$ and we are done after inserting $(p_j + d, q_j + d)$ into $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ (lines 3 and 4).

Else we can argue as in the proof of Lemma 8 to infer that O_y is the only a-heavy output interval of I_{s+1} starting before q_j, there are exactly $2a$ input intervals of I_{s+1} starting in O_y and that we call balance-up-to$(\langle p_z, q_z\rangle_{\mathsf{in}}, \langle p_y, q_y\rangle_{\mathsf{out}}, q_j)$ in line 12, which updates $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ to store I_t by Lemma 8. ◀

4.1.3 Running time analysis

▶ **Theorem 10.** *Given $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ for a disjoint interval sequence I of size k and an integer $a \geq 2$, we can compute $\mathcal{B}_a(I)$ in $\mathcal{O}(k + \frac{k}{a}\log k)$ time and $\mathcal{O}(1)$ additional space.*

Proof. As argued in Section 3.2, $\mathcal{T}_{\mathsf{in}}$ contains such an a-balanced disjoint interval sequence $\mathcal{B}_a(I)$ after performing Algorithm 1. Here, we iterate once over $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ in total. This takes $\mathcal{O}(k)$ time. Overall, we call balance-up-to (recursively and non-recursively) at most $\frac{k}{a-1}$ times, since by Theorem 6, the disjoint interval sequence is a-balanced after $\leq \frac{k}{a-1}$ balancing steps. In Algorithm 1, we need $\mathcal{O}(1)$ additional space.

In each of those calls of balance-up-to, we find $\langle p_{i+a}, q_{i+a}\rangle_{\mathsf{in}}$ ($\mathcal{O}(a)$ time), insert $(p_j + d, q_j + d)$ into $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$ ($\mathcal{O}(\log k)$ time) and possibly perform a predecessor- and a successor search over $\mathcal{T}_{\mathsf{out}}$ and $\mathcal{T}_{\mathsf{in}}$ ($\mathcal{O}(\log k)$ time), respectively. Overall, one call of balance-up-to needs $\mathcal{O}(a + \log k)$ time. We do not need the local variables in balance-up-to, once we recursively call balance-up-to, hence balance-up-to needs $\mathcal{O}(1)$ additional space. In total, this yields $\mathcal{O}(k + \frac{k}{a}(a + \log k)) = \mathcal{O}(k + \frac{k}{a}\log k)$ time and $\mathcal{O}(1)$ additional space. ◀

4.2 Practically optimized locate algorithm

Now we describe our practically optimized locate algorithm. At first, we introduce preparatory definitions and lemmas. Then, we discuss Nishimoto and Tabei's count algorithm [19] together with our practically optimized version of their locate algorithm.

▶ **Definition 11.** *Let $\mathcal{M}^{\mathsf{LF}}$ of size $r' \geq r$ be a move data structure (Corollary 7) built from the disjoint interval sequence $\mathcal{B}_a(I_{\mathsf{LF}})$, where $I_{\mathsf{LF}} = (l_1, \mathsf{LF}(l_1)), (l_2, \mathsf{LF}(l_2)), ..., (l_r, \mathsf{LF}(l_r))$. Let $\mathcal{M}^{\Phi}$ of size $r'' \geq r$ be a move data structure built from the disjoint interval sequence $\mathcal{B}_a(I_{\Phi})$, where $I_{\Phi} = (u_1, \Phi(u_1)), (u_2, \Phi(u_2)), ..., (u_r, \Phi(u_r))$. Let $L'[1..r']$ be a string containing the characters $L[\mathcal{M}^{\mathsf{LF}}_{\mathsf{p}}[1]], L[\mathcal{M}^{\mathsf{LF}}_{\mathsf{p}}[2]], ..., L[\mathcal{M}^{\mathsf{LF}}_{\mathsf{p}}[r']]$.*

Note that r' and r'' depend on the balancing parameter a. The total used space of these data structures is $\mathcal{O}(r)$, because $r', r'' \leq \frac{a}{a-1} r = \mathcal{O}(r)$ hold with Theorem 6.

▶ **Definition 12.** *Let $i \in [1, m]$, let $[b_i, e_i]$ be the suffix array interval of P_i, let $b'_i = \mathsf{LF}^{-1}(b_i)$, $e'_i = \mathsf{LF}^{-1}(e_i)$, let $\hat{b}_i$, $\hat{e}_i$, $\hat{b}'_i$ and $\hat{e}'_i$ be the indices of the input intervals in $\mathcal{M}^{\mathsf{LF}}$ containing b_i, e_i, b'_i and e'_i, respectively. Define $z(i) = \mathsf{SA}[e_i]$ (the lexicographically largest suffix of T starting with P_i), let $s(i,j) = z(j) + i - j$ and let $y(i) = \max\{j \in [1, m] \mid T_{s(i,j)}$ starts with $P_i\}$. Finally, define b_{m+1}, $\hat{b}_{m+1}$, e_{m+1} and $\hat{e}_{m+1}$ analogously, i.e, $b_{m+1} = 1$, $\hat{b}_{m+1} = 1$, $e_{m+1} = n$ and $\hat{e}_{m+1} = r'$.*

Let S be a string that occurs in T and let T_l be the lexicographically largest suffix of T among those starting with S. Then we call $T[l, l + |S|)$ the lexicographically largest occurrence of S in T. Intuitively, $P_{y(i)}$ is the shortest suffix of P s.t. the lexicographically largest occurrence of $P_{y(i)}$ in T is a suffix (in T) of the lexicographically largest occurrence of P_i in T.

▶ **Lemma 13.** *For $i \in [1, m]$, $T_{s(i,y(i))}$ is the lexicographically largest suffix of T starting with P_i, that is, it holds $s(i, y(i)) = z(i)$.*

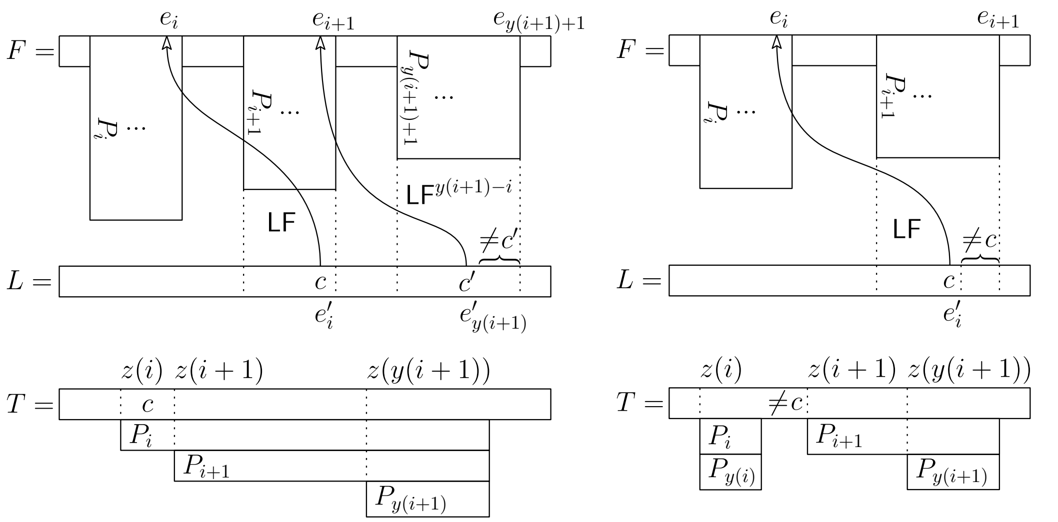

Figure 4 Illustrations of **Case** 1 (left) and **Case** 2 (right) of Lemma 15, where $c' = P[y(i+1)]$.

Proof. Suppose there was a suffix $T_w > T_{s(i,y(i))}$ starting with P_i. Then $T_{w+y(i)-i}$ starts with $P_{y(i)}$ and $T_{w+y(i)-i} > T_{z(y(i))}$, contradicting the definition of $T_{z(y(i))}$. ◀

▶ **Lemma 14.** *For $i \in [1, m]$, $p = \mathsf{SA}^{-1}[z(y(i)) + 1]$ is the end position of a BWT run.*

Proof. See Appendix A. ◀

The query $\mathsf{select}(T, c, i)$ returns the position of the i-th occurrence of c in T.

▶ **Lemma 15.** *Let $i \in [1, m]$. Given b_{i+1}, $\hat{b}_{i+1}$, e_{i+1} and $\hat{e}_{i+1}$ (and additionally $y(i+1)$ and $\hat{e}'_{y(i+1)}$, for $i < m$). If we augment L' with an $\mathcal{O}(\log\log_\omega \sigma)$ time and $\mathcal{O}(r')$ space* rank *data structure and an $\mathcal{O}(1)$ time and $\mathcal{O}(r')$ space* select *data structure, we can compute b_i, $\hat{b}_i$, e_i, $\hat{e}_i$, $y(i)$ and $\hat{e}'_{y(i)}$ in $\mathcal{O}(a + \log\log_\omega \sigma)$ time.*

Proof. Let $c = P[i]$. To obtain b_i and $\hat{b}_i$ from b_{i+1} and $\hat{b}_{i+1}$, we can use that $T_{\mathsf{SA}[b'_i]}$ is the lexicographically smallest suffix of T starting with P_{i+1} that is preceded by c, and therefore, b'_i and $\hat{b}'_i$ are the first occurrences of c at or after b_{i+1} and $\hat{b}'_{i+1}$ in L and L', respectively, allowing us to compute $\hat{b}'_i = \mathsf{select}(L', c, \mathsf{rank}(L', c, \hat{b}_{i+1} - 1) + 1)$, $b'_i = \mathcal{M}^{\mathsf{LF}}_{\mathsf{p}}[\hat{b}'_i]$ and $(b_i, \hat{b}_i) = \mathsf{move}(\mathcal{M}^{\mathsf{LF}}, b'_i, \hat{b}'_i)$. Analogously, we obtain e_i and $\hat{e}_i$ from e_{i+1} and $\hat{e}_{i+1}$ by $\hat{e}'_i = \mathsf{select}(L', c, \mathsf{rank}(L', c, \hat{e}_{i+1}))$, $e'_i = \mathcal{M}^{\mathsf{LF}}_{\mathsf{p}}[\hat{e}'_i + 1] - 1$ and $(e_i, \hat{e}_i) = \mathsf{move}(\mathcal{M}^{\mathsf{LF}}, e'_i, \hat{e}'_i)$. An implementation of the required rank-select data structure for L' can be found in [19] or Appendix B.1.

Now it remains to compute $y(i)$ and $\hat{e}'_{y(i)}$. If $i = m$, then $y(i) = m$, because P_m occurs at $z(m) \overset{Def.\ 12}{=} s(m, m)$, and therefore $\hat{e}'_{y(i)} = \hat{e}'_m$. Else, we have $i < m$. Here, we consider two cases (see Figure 4).

Case 1: $e'_i = e_{i+1}$. Then $T_{\mathsf{SA}[e_{i+1}]} \overset{Def.\ 12}{=} T_{z(i+1)}$ is preceded by c, hence P_i occurs at $z(i+1) - 1 \overset{Lem.\ 13}{=} s(i+1, y(i+1)) - 1 \overset{Def.\ 12}{=} s(i, y(i+1))$, hence (**i**) $y(i) \geq y(i+1)$. Let $j \in (y(i+1), m]$. Since P_{i+1} does not occur at $s(i+1, j)$, P_i does not occur at $s(i+1, j) - 1 \overset{Def.\ 12}{=} s(i, j)$, hence (**ii**) $y(i) \leq y(i+1)$. Combining (**i**) and (**ii**) yields $y(i) = y(i+1)$, and therefore $\hat{e}'_{y(i)} = \hat{e}'_{y(i+1)}$.

Case 2: $e'_i < e_{i+1}$. Then $T_{z(i+1)}$ is not preceded by c, hence P_i does not occur at $z(i+1)-1$. By induction, P_i also does not occur at $z(i+2)-2, z(i+3)-3, ..., z(m)+i-m$. More generally, P_i does not occur at $s(i,j)$, for each $j \in (i,m]$, hence **(i)** $y(i) \leq i$. Since P_i occurs at $z(i) \overset{Def.\ 12}{=} s(i,i)$, we get **(ii)** $y(i) \geq i$. Combining **(i)** and **(ii)** yields $y(i) = i$, and therefore $\hat{e}'_{y(i)} = \hat{e}'_i$.

In total, we perform two move queries on $\mathcal{M}^{\mathsf{LF}}$ ($\mathcal{O}(a)$ time) and at most two rank queries on L' ($\mathcal{O}(\log\log_\omega \sigma)$ time). The remaining computations take $\mathcal{O}(1)$ time, hence the running time bound follows. ◀

Note that iff $L'[\hat{b}_{i+1}] = c$ holds, then $b'_i = b_{i+1}$ and $\hat{b}'_i = \hat{b}_{i+1}$. Similarly, $L'[\hat{e}_{i+1}] = c \Leftrightarrow e'_i = e_{i+1} \wedge \hat{e}'_i = \hat{e}_{i+1}$. In practice, we use this to save up to two rank-select queries on L', which improves performance, especially if the text has a small alphabet. This optimization and the first part of Lemma 15, i.e, the computation of b_i, e_i, $\hat{b}_i$ and $\hat{e}_i$ is identical in the algorithms proposed by Nishimoto and Tabei [19].

▶ **Theorem 16.** *We can answer a count query in $\mathcal{O}(m(a + \log\log_\omega \sigma))$ time.*

Proof. We compute b_1, e_1, $\hat{b}_1$ and $\hat{e}_1$ by applying Lemma 15 m times starting with b_{m+1}, e_{m+1}, $\hat{b}_{m+1}$ and $\hat{e}_{m+1}$ and return the length $|[b_1, e_1]|$ of the suffix array interval of P. ◀

Note that we do not need $y(1)$ and $\hat{e}'_{y(1)}$ to answer a count query. Those values are only necessary for answering a locate query, which we will discuss next.

▶ **Definition 17.** *For $i \in [1, r']$, let $p_i = \mathcal{M}^{\mathsf{LF}}_{\mathsf{p}}[i+1] - 1$. If p_i is the end position of a BWT run, then there exists an output interval in I_Φ (and therefore also in $\mathcal{M}^\Phi$) that starts with $\mathsf{SA}[p_i]$. Let x_i be the index of this output interval of $\mathcal{M}^\Phi$, i.e, $x_i \in [1, r'']$ s.t. $\mathcal{M}^\Phi_{\mathsf{q}}[x_i] = \mathsf{SA}[p_i]$. Finally, let $\mathsf{SA}_\Phi[1..r']$ be an array, where*

$$\mathsf{SA}_\Phi[i] = \begin{cases} x_i & p_i \textit{ is the end position of a BWT run,} \\ \perp & \textit{else.} \end{cases}$$

▶ **Theorem 18.** *We can answer a locate query in $\mathcal{O}(m(a + \log\log_\omega \sigma) + occ \cdot a)$ time.*

Proof. As in Theorem 16, we compute b_1, e_1, $\hat{b}_1$ and $\hat{e}_1$ and additionally $y(1)$ and $\hat{e}'_{y(1)}$ by applying Lemma 15 m times starting with b_{m+1}, e_{m+1}, $\hat{b}_{m+1}$ and $\hat{e}_{m+1}$. This takes $\mathcal{O}(m(a + \log\log_\omega \sigma))$ time. Then, we compute

$$\begin{aligned} s = \mathsf{SA}[e_1] &\overset{Lem.\ 13}{=} s(1, y(1)) \overset{Def.\ 12}{=} z(y(1)) + 1 - y(1) \overset{Def.\ 12}{=} \mathsf{SA}[e_{y(1)}] + 1 - y(1) \\ &\overset{Def.\ 12}{=} \mathsf{SA}[e'_{y(1)}] - y(1) \overset{\textbf{(i)}}{=} \mathsf{SA}[\mathcal{M}^{\mathsf{LF}}_{\mathsf{p}}[\hat{e}'_{y(1)} + 1] - 1] - y(1) \overset{\textbf{(ii)}}{=} \mathcal{M}^\Phi_{\mathsf{q}}[\mathsf{SA}_\Phi[\hat{e}'_{y(1)}]] - y(1) \end{aligned}$$

and report s. Lemma 14 yields, that $e'_{y(1)} \overset{Def.\ 12}{=} \mathsf{LF}^{-1}(e_{y(1)}) \overset{Def.\ 12}{=} \mathsf{SA}^{-1}[z(y(1)) + 1]$ is the end position of a BWT run. Since the BWT run end positions are a subset of the input interval end positions of $\mathcal{M}^{\mathsf{LF}}$, $e'_{y(1)}$ is also the end position of the $\hat{e}'_{y(1)}$-th input interval of $\mathcal{M}^{\mathsf{LF}}$, hence **(i)** holds and **(ii)** follows with Definition 17. If $b_1 = e_1$, then we are done.

Else, we compute the remaining occurrences $\{\mathsf{SA}[e_1 - 1], \mathsf{SA}[e_1 - 2], ..., \mathsf{SA}[b_1]\} = \{\Phi^1(s), \Phi^2(s), ..., \Phi^{e_1 - b_1}(s)\}$ by consecutively performing $e_1 - b_1$ move queries on $\mathcal{M}^\Phi$ starting with $(s, \hat{s})$ and reporting s after each move query, where $\hat{s}$ is the index of the input interval in $\mathcal{M}^\Phi$ containing s. This takes $\mathcal{O}(occ \cdot a)$ additional time. To compute $\hat{s}$, we at first compute the index $\hat{x} = \mathcal{M}^\Phi_{\mathsf{idx}}[\mathsf{SA}_\Phi[\hat{e}'_{y(1)}]]$ of the input interval in $\mathcal{M}^\Phi$ containing $\mathsf{SA}[e'_{y(1)}]$. Again, we can do this by Definition 17, because $e'_{y(1)}$ is the end position of both a BWT run and the $\hat{e}'_{y(1)}$-th input interval of $\mathcal{M}^{\mathsf{LF}}$. Then, we find $\hat{s}$ in $\mathcal{O}(\log m)$ time with an exponential search to the left over the input interval starting positions of $\mathcal{M}^{\Phi}$ starting at $\hat{x}$. We can do this, because $\hat{x} - \hat{s} \overset{\textbf{(iii)}}{\leq} \mathsf{SA}[e'_{y(1)}] - \mathsf{SA}[e_1] \overset{\textit{Def. 12, Lem. 13}}{=} z(y(1)) + 1 - s(1, y(1)) \overset{\textit{Def. 12}}{=} y(1) \overset{\textit{Def. 12}}{\leq} m$, where **(iii)** holds, because an input interval of $\mathcal{M}^{\Phi}$ has at least size one. ◀

Comparing Move-r with OptBWTR

Move-r consists of the data structures $\mathcal{M}^{\mathsf{LF}}$, $\mathcal{M}^{\Phi}$, L' (see Definition 11), a rank-select data structure $RS_{L'}$ for L' (see Appendix B.1) and SA_{Φ} (see Definition 17). Overall, Move-r needs $\mathcal{O}(r)$ space, because $r', r'' = \mathcal{O}(r)$ hold with Theorem 6.

Now we discuss the differences between OptBWTR and Move-r. OptBWTR implements Φ^{-1} instead of Φ, which results in symmetric algorithms and data structures. To compute the initial suffix array value during a locate query, OptBWTR stores two arrays $\mathsf{SA}_{\mathsf{s}}[1..r']$ and $\mathsf{SA}_{\mathsf{idx}}[1..r']$ instead of SA_{Φ}, where $\mathsf{SA}_{\mathsf{s}}[i] = \mathsf{SA}[\mathcal{M}^{\mathsf{LF}}_{\mathsf{p}}[i]]$ and $\mathsf{SA}_{\mathsf{idx}}[i]$ stores the index of the input interval in $\mathcal{M}^{\Phi}$ containing $\mathsf{SA}_{\mathsf{s}}[i]$. With Move-r, we access SA_{Φ} only once and $\mathcal{M}^{\Phi}_{\mathsf{p}}$ at most $\log m$ times to compute the initial suffix array value during a locate query. This saves up to $3m$ cache misses compared with OptBWTR, because there, $\mathcal{M}^{\Phi}_{\mathsf{p}}$, SA_{s} and $\mathsf{SA}_{\mathsf{idx}}$ are randomly accessed in potentially each iteration of the backward search. Another difference is that Move-r uses a practically optimized implementation of $RS_{L'}$ (see Appendix B.1). Finally, Move-r encorporates further practical optimizations (see Appendix B.2 and Appendix B.3).

5 Experimental Evaluation

We implemented the algorithms described in Section 4 in `C++20`. We implemented a practically optimized version the balancing algorithm described in Section 4.1. For $\mathcal{T}_{\mathsf{in}}$ and $\mathcal{T}_{\mathsf{out}}$, we used the B-tree implementation from the `abseil-cpp` library. For sorting, we used the in-place sample sort implementation `ips4o` [1]. We used the uncompressed bit vector- and sd-array implementations in the SDSL [12] to implement $RS_{L'}$. To measure peak memory consumption, we used `malloc_count`. Links to all software used can be found in our GitHub repository.

Now we discuss the tested indexes. Some of them use `Big-BWT` [15], which constructs the so-called *prefix free parsing* (PFP) [4] of T to build the BWT and suffix array samples. This approach reduces the working space needed to construct an r-index from $\mathcal{O}(n)$ to $\mathcal{O}(|\mathsf{PFP}|)$ words, where $|\mathsf{PFP}|$ is the sum of the lengths of all dictionary phrases and the number of all phrases in the factorization of the PFP of T. `grlBWT` [9] is a BWT construction algorithm for string collections that uses string compression. For most inputs, it is currently the fastest and most memory-efficient algorithm. However, it does not support the computation of suffix-array samples, which are necessary for efficiently answering locate queries.

- `move-r`: Our implementation. It uses `Big-BWT` build the BWT and suffix array samples. In preliminary experiments, we observed that the balancing process can drastically increase query throughput and determined $a = 8$ to be the optimal trade-off between index size and performance.
- `r-index`: [11] The original implementation of the r-index (adjusted to use `Big-BWT`).
- `online-rlbwt`: [2] A refined version of the dynamic RLBWT implementation from [22], which additionally supports answering locate queries.
- `rcomp-glfig`: [18] A dynamic r-index developed by the inventors of the move data structure. It uses their so-called *divided BWT* (DBWT) and *grouped* LF-*interval graph* (GLFIG) representations of the BWT, which maintain the stricter so-called α-balancedness property, where $\alpha \geq 2$ is an integer. The grouping parameter g has been set to 16.

Table 1 Statistics of the tested texts (r' and r'' calculated for $a = 2$). N denotes the number of queried patterns, m is the pattern length and $\overline{occ}$ is the average number of occurrences per pattern.

text	size [GB]	σ	n/r	r'/r	r''/r	N	m	$\overline{occ}$
einstein.en.txt	0.47	139	1611.18	1.23	1.49	100000	800	748.82
						10000	7	74237.84
sars2	84.19	80	686.57	1.38	1.06	1000	5000	7290.24
						50	50	524838.82
dewiki	68.72	210	345.80	1.23	1.35	20000	320	313.49
						130	10	216783.43
chr19	58.57	52	272.20	1.06	2.00	150	50000	61806.96
						200	80	719590.32
english	2.21	239	3.36	1.19	1.20	30000	35	34.60
						150	7	98864.59

For reasons of completeness, we also include the following data structures in our test, although they do not support locate queries. They should thus not be directly compared with the other indexes.

- `r-index-f`: [5] An r-index that also uses a move data structure to compute LF. However, it does not perform a balancing algorithm on I_{LF}. It uses `Big-BWT` and `pfp-thresholds`, to build the BWT, and has been optimized to reduce the index size. It does not support answering locate queries. We used the variant `lookup-bv` as recommended by the authors.
- `block-rlbwt-2`: [8] An index that splits L into blocks of size $b = 2^{11}$ and applies run-length encoding to those. $\mathsf{rank}(L, c, i)$ is computed by looking up the number of occurrences of c before the $\lfloor i/b \rfloor$-th block and scanning over it up to position i. Runs exceeding length $2^{16-\lceil \log_2 \sigma \rceil}$ are split s.t. one run can be encoded using two bytes. It uses `grlBWT` to build the BWT and does not support answering locate queries.
- `block-rlbwt-v`: [8] Uses the same approach as `block-rlbwt-2`, but avoids splitting runs by using $\mathcal{O}(\log l + \log \sigma)$ bytes to encode a run of length l, and uses $b = 2^{14}$.
- `block-rlbwt-r`: [8] Uses the same approach as `block-rlbwt-2`, but instead splits L into blocks of $b' = 32$ runs. To compute $\mathsf{rank}(L, c, i)$, the block containing i is found using a heap ordered B$^+$-tree of the block starting positions.

We compared all indexes using our tool `move-r-bench`, which is included in our GitHub repository, on a system with two AMD EPYC 7452 CPUs (32/64x 2.35-3.35GHz, 2/16/128MB L1/2/3 cache) and 1TB of 3200 MT/s DDR4 RAM using the GCC 9.4.0 compiler and the compile flags `"-march=native -DNDEBUG -Ofast"` on Ubuntu 18.04.6. Table 1 shows the tested texts. einstein.en.txt and english are part of the Pizza&Chili Corpus. dewiki is a highly repetitive text that has been handcrafted from German Wikipedia entries. chr19 consists of concatenated human chromosome 19 haplotypes, and sars2 is a collection of Sars-Cov-2 genomes, both of which were crafted out of datasets from the National Center for Biotechnology Information (NCBI) database. Links to all texts can be found in our GitHub repository.

5.1 Construction performance

Figure 5 shows index construction performance (time and peak memory consumption). Commonly, the `block-rlbwt` indexes can be constructed the fastest and while using the least space (except with einstein.en.txt). This is because they use `grlBWT` to build the BWT.

With repetitive texts, index construction time and peak memory usage are dominated by the construction of the BWT. In this case, any static r-index can easily be adapted to instead use `grlBWT` and achieve similar construction performance with the limitation that it then only supports count queries. Compared with the closest other index supporting locate queries, `move-r` can be constructed 0.9–2 (typically 2) times as fast while requiring 1/3–1 (typically 1/2) times as much memory. In some cases, `move-r` even competes with the dynamic indexes regarding memory usage (see chr19, dewiki and english). Constructing static indexes is 2–20 times as fast and requires 1–10 times as much space than constructing dynamic indexes. Comparing the two dynamic indexes, `rcomp-glfig`'s construction consumes 2–3 times as much memory as that of `online-rlbwt` while sometimes one and sometimes the other takes (at most 60%) longer.

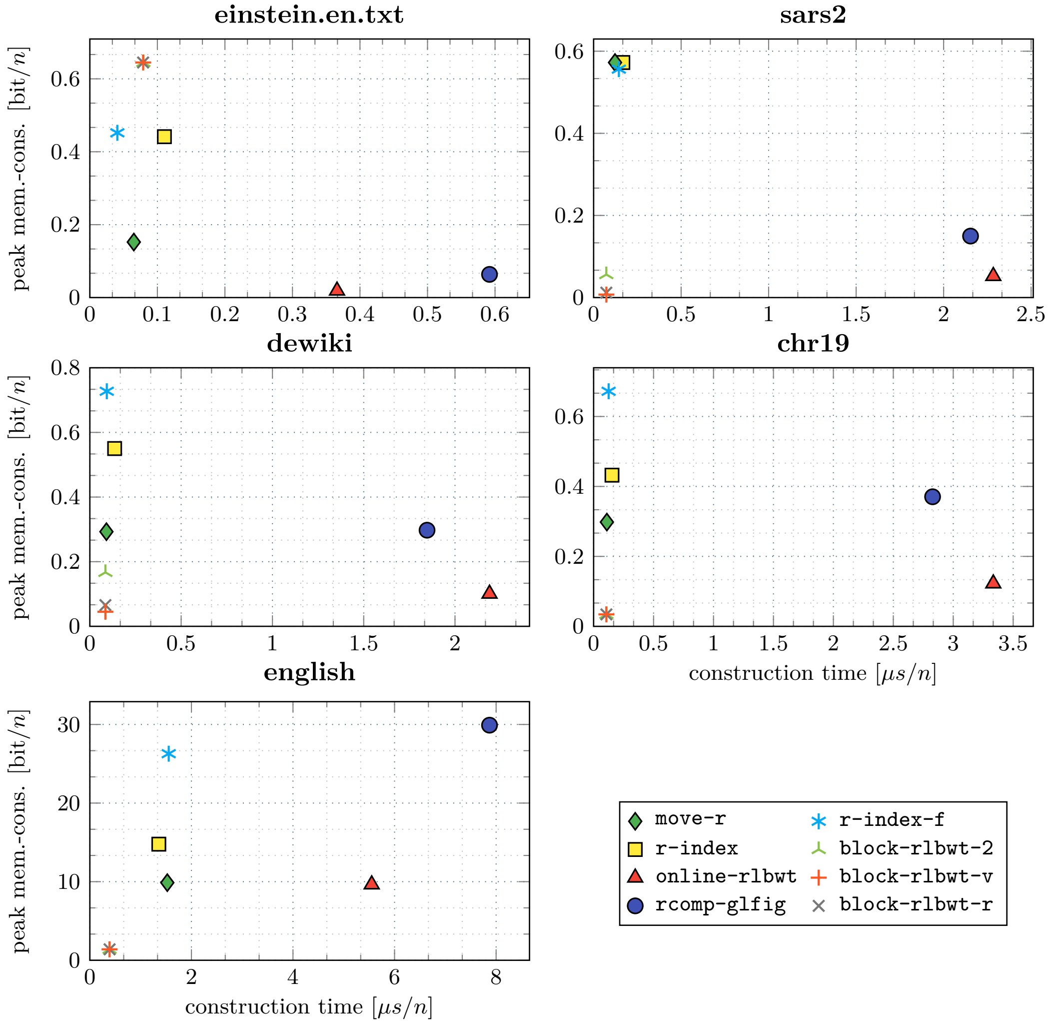

Figure 5 Peak memory consumption during the index construction versus index construction time. The indexes in the left legend column (solid marks) support answering locate queries, while the others do not.

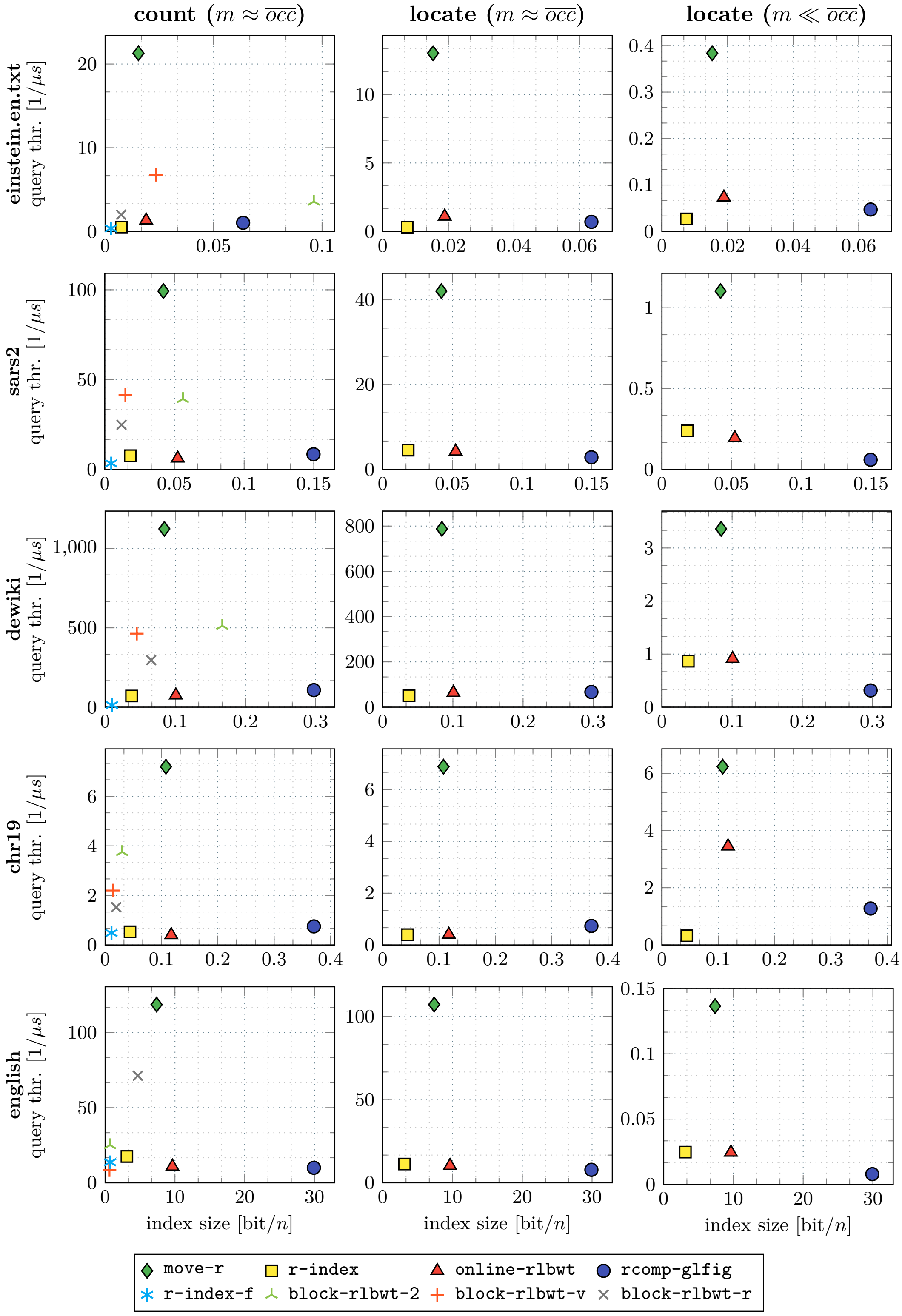

Figure 6 Query throughput versus index size. The indexes in the upper legend row (solid marks) support answering locate queries, while the others do not.

5.2 Query performance

For each text, we generated two sets of query patterns (two lines per file in Table 1) using our tool `move-r-patterns`. The tool, generated patterns and scripts for replicating our experiments are included in our GitHub repository. We chose the patterns in the first set s.t. $\overline{occ} \approx m$. This implies that when locating those patterns, we measure a blend of LF-, Φ- and rank-select queries on L', since we perform $\overline{occ} \approx m$ Φ queries, $2m$ LF queries and at most $2m$ rank-select queries on L'. The patterns in the second set were chosen s.t. $\overline{occ} \approx 10^5 m$. When locating those patterns, we practically only measure Φ query performance. To measure count performance, we used the first set of patterns.

Figure 6 shows query performance versus index size. Out of the indexes without locate support, different `block-rlbwt` indexes provide the best trade-off between query performance and index size, depending on the repetitiveness and alphabet size of the text (see [8] for a more detailed discussion). `r-index-f` is consistently the smallest index, but also achieves low query throughput. Out of the dynamic indexes, `online-rlbwt` clearly performs better, because `rcomp-glfig` is consistently 3–4 times as large as it and often achieves a lower query throughput. `r-index` usually provides query performance similar to `online-rlbwt` while being only 1/3 as large. `move-r` outperforms `r-index` by factors between 3 and 35 (typically 15) while being 2–2.5 times as large. Comparing `move-r` with the fastest `block-rlbwt` index, respectively, `move-r` is 2–3 (typically 2) times as fast and 0.8–6 (typically 2) times as large. However, `move-r` can also be constructed without locate support, which roughly halves its size. This lessens (by a factor of 2) and sometimes cancels out the index size advantage of the respective fastest `block-rlbwt` index.

6 Conclusion

Overall, we have shown that the move data structure speeds up the r-index while causing an acceptable space increase and can be constructed efficiently. Regarding query throughput, Move-r outperforms the fastest other r-index supporting locate queries ([11] or [2]) by factors between 2 and 35 (typically 15) while being 0.8–2.5 (typically 2) times as large. Move-r can be constructed 0.9–2 (typically 2) times as fast while consuming 1/3–1 (typically 1/2) times as much memory. Compared with the fastest r-index supporting only count queries, Move-r achieves 2–3 (typically 2) times better query throughput while being 0.4–3 (typically 1) times as large.

References

1 Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms. *ACM Transactions on Parallel Computing*, 9(1):2:1–2:62, 2022.

2 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.

3 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):31:1–31:21, 2015.

4 Christina Boucher, Travis Gagie, Alan Kuhnle, and Giovanni Manzini. Prefix-free parsing for building big bwts. In *18th International Workshop on Algorithms in Bioinformatics WABI*, pages 2:1–2:16, 2018.

5 Nathaniel K. Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. In *20th International Symposium on Experimental Algorithms SEA*, pages 16:1–16:16, 2022.

6 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. In *Technical Report DEC*, volume 124, 1994.
7 David Richard Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, CAN, 1998.
8 Diego Díaz-Domínguez, Saska Dönges, Simon J. Puglisi, and Leena Salmela. Simple runs-bounded fm-index designs are fast. In *21st International Symposium on Experimental Algorithms SEA*, pages 7:1–7:16, 2023.
9 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*, pages 29:1–29:18, 2022.
10 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
11 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *Journal of the ACM*, 67(1):2:1–2:54, 2020.
12 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms SEA*, pages 326–337, 2014.
13 Torben Hagerup. Sorting and searching on the word RAM. In *15th Annual Symposium on Theoretical Aspects of Computer Science STACS*, Lecture Notes in Computer Science, pages 366–398, 1998.
14 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.
15 Taher Mun, Alan Kuhnle, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Matching reads to many genomes with the r-index. *Journal of Computational Biology*, 27(4):514–518, 2020.
16 Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.
17 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Computing Surveys*, 54(2):29:1–29:31, 2022.
18 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*, pages 99:1–99:20, 2022.
19 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on bwt-runs compressed indexes. In *48th International Colloquium on Automata, Languages, and Programming ICALP*, pages 101:1–101:15, 2021.
20 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *9th Workshop on Algorithm Engineering and Experiments ALENEX*, pages 60–70, 2007.
21 Anna Pagh, Rasmus Pagh, and Milan Ruzic. Linear probing with 5-wise independence. *SIAM Review*, 53(3):547–558, 2011.
22 Nicola Prezza. A framework of dynamic data structures for string processing. In *16th International Symposium on Experimental Algorithms SEA*, pages 11:1–11:15, 2017.
23 Sebastiano Vigna. Broadword implementation of rank/select queries. In *7th International Workshop on Experimental Algorithms WEA*, pages 154–168, 2008.

A Proof of Lemma 14

▶ **Lemma 14.** *For $i \in [1,m]$, $p = \mathsf{SA}^{-1}[z(y(i)) + 1]$ is the end position of a BWT run.*

Proof. Let $p' = \mathsf{SA}^{-1}[z(y(i) + 1)]$. By the definition of $P_{y(i)}$, $T_{z(y(i)+1)-1}$ does not start with $P_{y(i)}$, hence $L[p'] = T[z(y(i) + 1) - 1] \neq P[y(i)]$. The definition of $T_{z(y(i)+1)}$ implies $T_{z(y(i))+1} < T_{z(y(i)+1)}$ and therefore $p < p'$. Now suppose p was not the end position of a BWT run. Since $L[p + 1] = L[p] = P[y(i)]$ implies $p + 1 < p'$, $T_{\mathsf{SA}[p+1]}$ starts with

$P_{y(i)+1}$ and $T_{\mathsf{SA}[\mathsf{LF}(p+1)]}$ starts with $P_{y(i)}$. However, because $\mathsf{LF}(p) < \mathsf{LF}(p+1)$ follows with the LF-property, $T_{\mathsf{SA}[\mathsf{LF}(p+1)]}$ is lexicographically larger than $T_{z(y(i))}$, which contradicts the definition of $T_{z(y(i))}$, hence the claim is correct. ◀

B More practical optimizations

B.1 Practically optimized rank-select data structure

Nishimoto and Tabei proposed the following data structures to implement $RS_{L'}$:

- $RS_{\text{map}}[1..\sigma]$: deterministic dictionary [21] storing the order-preserving mapping function from Σ to the effective alphabet Σ' of Σ
- RS_{rank}: wavelet tree [3] of L'
- RS_{select} of size r', where $RS_{\mathsf{select}}[c][i] = \mathsf{select}(L', c, i)$ for $c \in \Sigma'$ and $i \in [1, |\text{Occ}(c, L')|]$

With RS_{rank}, we can answer rank queries on L' in $\mathcal{O}(\log\log_\omega \sigma)$ time. With RS_{map} and RS_{select}, we can answer select queries on L' in $\mathcal{O}(1)$ time. With the move data structure $\mathcal{M}^{\mathsf{LF}}$, the asymptotic runtime of the count algorithm is bounded by the runtime $\mathcal{O}(m \log\log_\omega \sigma)$ (for $a = \mathcal{O}(1)$) of the rank queries on the wavelet-tree of L'. This is also the case in the locate alorithm, if $occ = \mathcal{O}(m \log\log_\omega \sigma)$ (for $a = \mathcal{O}(1)$).

We instead store σ bit-vectors of size r' each in $RS_{L'}[1..\sigma][1..r']$, where $RS_{L'}[c][i] = 1 \Leftrightarrow L'[i] = c$. Then we can answer $\mathsf{rank}(L', c, i)$ by computing $RS_{L'}[c].\mathsf{rank}_1(i)$. Similarly, we compute $RS_{L'}[c].\mathsf{select}_1(i)$ to answer $\mathsf{select}(L', c, i)$.

If a character $c \in \Sigma$ occurs at least $r'/10$ times in L', we implement $RS_{L'}[c][1..r']$ with an uncompressed bit vector and augment it with $\mathcal{O}(1)$ time and $o(r')$ space rank_1- [23] and select_1 [7] data structures. Else, we use an sarray [20] to implement $RS_{L'}[c][1..r']$. This results in $\mathcal{O}(\log \frac{r'}{n_c} + \log^4 n_c / \log r')$ time to answer a rank query on L' and $\mathcal{O}(\log^4 n_c / \log r')$ time to answer a select query on L', where $c \in \Sigma$ and n_c denotes the number of occurrences of c in L'. Overall, we need $\mathcal{O}(\sum_{i=1}^{m}(a + \log \frac{r'}{n_{P[i]}} + \log^4 n_{P[i]} / \log r')) = \mathcal{O}(m(a + \log^3 r))$ time to answer a count query, because $1 \leq n_c < r' \, \forall c \in \Sigma$ and $r' = \mathcal{O}(r)$, where $n_{P[i]}$ denotes the number of occurrences of $P[i]$ in L'. This is asymptotically worse than the $\mathcal{O}(m(a + \log\log_\omega \sigma))$ time that we get when using a wavelet-tree of L'. In practice, our rank-select data structure reduces the running time of count- and locate queries (for $occ \lessapprox m$) by a factor up to 3.

Since there are at most 9 characters $c \in \Sigma$ occurring at least $r'/10$ times in L' (because \$ occurs exactly once in L'), there are at most 9 uncompressed bit vectors in $RS_{L'}$, which need $9r' + o(r')$ bits. The size of the sarrays in $RS_{L'}$ is bounded by $\sum_{c \in \Sigma}(n_c \log(\frac{r'}{n_c}) + 2n_c + o(n_c)) = \sum_{c \in \Sigma}(n_c \log(\frac{r'}{n_c})) + 2r' + o(r') = r'\mathcal{H}_0(L') + 2r' + o(r')$ bits, which is asymptotically not larger than a huffman-shaped wavelet-tree of L', which needs $r'\mathcal{H}_0(L') + o(r'\mathcal{H}_0(L') + 1) + \sigma\omega$ bits (see [16]), where $\mathcal{H}_0(L')$ is the Zeroth Order Entropy of L'. In practice, $RS_{L'}$ is roughly 2 times as large as a huffman-shaped wavelet-tree of L'.

B.2 Reducing index construction time in practice

In [19], they showed how to construct $\mathcal{M}_{\mathsf{idx}}$ in $\mathcal{O}(k' \log k')$ time using a binary search over $\mathcal{M}_{\mathsf{p}}[1..k']$ for each entry in $\mathcal{M}_{\mathsf{idx}}[1..k']$. To speed up the construction in practice, we compute π for $\mathcal{B}_a(I)$ and use it to iterate over the output intervals of $\mathcal{B}_a(I)$ in $\mathcal{O}(k')$ time. We simultaneously iterate over the input intervals to keep track of the index i of the input interval containing the starting position of the current (j-th) output interval. Hence, we can write $\mathcal{M}_{\mathsf{idx}}[\pi[j]] \leftarrow i$ for each j-th output interval. Since the construction of π takes $\mathcal{O}(k' \log k')$ time, this method still results in the same theoretical time to construct $\mathcal{M}_{\mathsf{idx}}$, but is much faster in practice.

We can apply a similar method to build SA_Φ. We at first build the array $\mathsf{SA}'_\mathsf{s}[1..r']$, where $\mathsf{SA}'_\mathsf{s}[i] = \mathsf{SA}[\mathcal{M}^{\mathsf{LF}}_\mathsf{p}[i+1]-1]$, if p_i is the end position of a BWT run and $\mathsf{SA}'_\mathsf{s}[i] = \infty$, else. Then, we build the permutation $\pi'[1..r']$ storing the order of the values in SA'_s in $\mathcal{O}(r' \log r')$ time. To iterate over the output intervals of $\mathcal{M}^\Phi$, we can reuse π_Φ from the construction of $\mathcal{M}^\Phi_{\mathsf{idx}}$, i.e, we can build SA_Φ by simultaneously iterating over $\mathcal{M}^\Phi_\mathsf{q}[\pi_\Phi[1]], \mathcal{M}^\Phi_\mathsf{q}[\pi_\Phi[2]], ..., \mathcal{M}^\Phi_\mathsf{q}[\pi_\Phi[r'']]$ and $\mathsf{SA}'_\mathsf{s}[\pi'[1]], \mathsf{SA}'_\mathsf{s}[\pi'[2]], ..., \mathsf{SA}'_\mathsf{s}[\pi'[r]]$ in $\mathcal{O}(r + r'') = \mathcal{O}(r)$ time and setting $\mathsf{SA}_\Phi[\pi'[i]] \leftarrow \perp$ for $i \in (r, r']$ in $\mathcal{O}(r' - r) = \mathcal{O}(r)$ time.

B.3 Reducing index size in practice

We reduced the size of the move data structure by using the following tricks. Instead of storing $\mathcal{M}_\mathsf{q}$, we only store the offset of each output interval starting position in the input interval containing it, i.e, we store the arrays $\mathcal{M}_\mathsf{p}[1..k]$, $\mathcal{M}_{\mathsf{offs}}[1..k]$ and $\mathcal{M}_{\mathsf{idx}}[1..k]$, where $\mathcal{M}_{\mathsf{offs}}[i] = q_i - \mathcal{M}_\mathsf{p}[\mathcal{M}_{\mathsf{idx}}[i]]$, for $i \in [1, k]$. This allows us to compute q_i with one random access to $\mathcal{M}_\mathsf{p}$ at the position $\mathcal{M}_{\mathsf{idx}}[i]$, which is irrelevant when evaluating a move query, because when computing $\mathsf{move}(\mathcal{M}, i, x) = (i', x')$, we start scanning over $\mathcal{M}_\mathsf{p}$ at the position $\mathcal{M}_{\mathsf{idx}}[x]$ to find x'.

The length of the longest input interval is an upper bound of any value in $\mathcal{M}_{\mathsf{offs}}$. This implies that we can store $\mathcal{M}_{\mathsf{offs}}$ with the word width ω_{offs} by limiting the length of each input interval to $2^{\omega_{\mathsf{offs}}}$. We can do this by iterating over the input intervals from left to right and splitting each input interval considered (and its corresponding output interval) at the offset $2^{\omega_{\mathsf{offs}}}$ if it is longer than $2^{\omega_{\mathsf{offs}}}$. We split the input intervals before performing the balancing algorithm, because splitting an input interval can make the output interval containing the split position a-heavy. Since we perform $\leq \frac{n}{2^{\omega_{\mathsf{offs}}}}$ splits, the number of input- and output intervals increases by a factor $\leq 1 + \frac{n}{k 2^{\omega_{\mathsf{offs}}}}$. We choose $\omega_{\mathsf{offs}} = \min\{\omega \in \{8, 16, 24, 32, 40\} : \frac{n}{k 2^\omega} \leq \epsilon\}$ to bound the overall size of the resulting move data structure to $\mathcal{O}((1+\epsilon)\frac{a}{a-1}k)$, for some $\epsilon > 0$. This suffices, if we assume $n \leq 2^{40}$, that is if the text file is smaller than $\approx$ 1TB. In practice, $\epsilon = \frac{1}{8}$ turns out to be reasonable trade-off.

We also store $\mathcal{M}_\mathsf{p}$ and $\mathcal{M}_{\mathsf{idx}}$ with the minimum possible word widths $\omega_\mathsf{p} = \min\{\omega \in \{8, 16, 24, 32, 40\} : n \leq 2^\omega\}$ and $\omega_{\mathsf{idx}} = \min\{\omega \in \{8, 16, 24, 32, 40\} : k' \leq 2^\omega\}$, respectively. Finally, we store $\mathcal{M}_\mathsf{p}$, $\mathcal{M}_{\mathsf{offs}}$ and $\mathcal{M}_{\mathsf{idx}}$ interleaved with each other, to reduce the number of cache misses when performing move queries.

To further reduce the number of cache misses, we store L' interleaved with the arrays of $\mathcal{M}^{\mathsf{LF}}$. Since $\mathsf{SA}_\Phi[i] \in [1, r'']$, we can store SA_Φ with the same word width ω_{idx} as $\mathcal{M}^\Phi_{\mathsf{idx}}$.