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 incorporates 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 compressibility 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.

Keywords and phrases
Compressed Text Index, Burrows-Wheeler Transform

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 $O(r)$ 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 $O(m \log \log_w(\sigma + n/r))$ time, and locate queries (listing all matching positions) in additional $O(\text{occ} \log \log_w(n/r))$ time, where $n$ is the total text length, occ is the number of matches, $\sigma$ 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 $O(1)$ time. The move data structure uses the property that two important functions (LF and $\Phi$, 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 $O(1)$ time, while still using only $O(r)$ space.
1:2 Move-r: Optimizing the r-index

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 \( O(1) \) time over the \( O(\log \log(n/r)) \) 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 \(\{i, \ldots, j\}\). For convenience, we use the following notations: \([i, j] = [i, j + 1] = (i - 1, j + 1)\). Given a set \(S \subset U\), we define the predecessor of \(i \in U\) in \(S\) as \(\text{pred}(i) = \max\{j | j \in S \land j < i\}\). Similarly, we define the successor of \(i \in U\) in \(S\) as \(\text{succ}(i) = \min\{j | j \in S \land j > i\}\). We assume that all described algorithms work in the word-RAM model with word size \(w\) [13].

An alphabet \(\Sigma\) 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 \(\Sigma\). In case that \(|T| = 0\), \(T\) is the empty string \(\epsilon\). We can access the \(i\)-th symbol of \(T\) with \(T[i]\). A substring \(T[i, j]\) is defined by \(T[i] \cdot T[i + 1] \cdots \cdot T[j]\). 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 \(c_1 < c_2 < \ldots < c_\sigma\) on the alphabet \(\Sigma = \{c_1, c_2, \ldots, c_\sigma\}\). The lexicographic order of strings is then defined by \(T < T' \iff T\) is a proper prefix of \(T' \lor \exists i : T[1, i] = T'[1, i] \land T[i + 1] < T'[i + 1]\). We are also interested in how often a character \(c\) appears in a string \(T[1, i]\), which we denote by \(\text{rank}(T, c, i)\).

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 \(\Sigma\). This allows us to simplify our algorithms.

The suffix array [14] \(\text{SA}\) of a string \(T\) consists of the starting positions of all suffixes of \(T\) in their lexicographic order, i.e., \(T_{\text{SA}[1]} < T_{\text{SA}[2]} < \ldots < T_{\text{SA}[\sigma]}\). 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) \cite{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 $L = L_1 \cdot L_2 \cdot \cdots \cdot L_r$. 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 $LF(i) = j \iff SA[j] = SA[i] - 1$ if $SA[i] > 1$, and $LF(i) = 1$ if $SA[i] = 1$. The LF property describes that the order of same symbols is identical in $F$ and $L$, i.e., $L[i] = L[j]$ and $i < j \Rightarrow LF(i) < LF(j)$. 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, $LF(i) = C[L[i]] + \text{rank}(L, L[i], i)$ 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 $O(r)$ space:

$\blacktriangleright$ Lemma 1. Let $T$ be a string of length $n$ and $L$ be its BWT. Let $l_1 < l_2 < \ldots < l_r$ be the starting positions of each run in $L$ and let $l_{r+1} = n + 1$. Then $LF(i) = LF(l_x) + (i - l_x)$ 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 \cite{10}. We search for $P$ by iterating from right to left over $P$ and determining in iteration $i$ (implicitly) the suffix array interval $[b, c]$ that is prefixed by $P[m - i + 1, m]$. We initialize $[b, c]$ 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 $[b, c]$ by using rank queries. So, we set $b = C[c] + \text{rank}(L, c, b - 1) + 1$ and $c = C[c] + \text{rank}(L, c, c)$. Having read $P$ completely, the output $|[b, c]| = n$ is the number of occurrences of $P$ in $T$. If $|[b, c]| = 0$, there are no occurrences of $P$ in $T$. If the suffix array is at hand, we can also answer locate queries by outputing 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 $\Phi$, where $\Phi(SA[i]) = SA[i - 1]$ for $i > 1$ and $\Phi(SA[1]) = SA[n]$. The following property holds, which allows us to implement $\Phi$ in $O(r)$ space – note the similarity to Lemma 1.

$\blacktriangleright$ Lemma 2 (\cite{11}, lemma 3.5). Let $\{u_1, u_2, \ldots, u_{r+1}\} = \{SA[l_1], SA[l_2], \ldots, SA[l_r], n + 1\}$ and $u_1 < u_2 < \ldots < u_{r+1} = n + 1$. 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 \cite{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 $\Phi$ 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 \cite{19} and describe how to efficiently answer queries on interval sequences.
3.1 Disjoint Interval Sequence

We first formally define disjoint interval sequences.

Definition 3. Let I = (p₁, q₁), (p₂, q₂), ..., (pₖ, qₖ) be a sequence of pairs of elements in the range [1, n], where q₁, q₂, ..., qₖ are pairwise distinct, let π be the permutation of [1, k] s.t. 1 = q_{π[1]} < q_{π[2]} < ... < q_{π[k]} ≤ n, let p_{k+1} = n + 1 = q_{k+1}, and let dᵢ = p_{i+1} − pᵢ for i ∈ [1, k]. Then, I is called a disjoint interval sequence if 1 = p₁ < p₂ < ... < pₖ < p_{k+1} and d_{π[i]} = q_{π[i+1]} − q_{π[i]} for all i ∈ [1, k].

A disjoint interval sequence defines a sequence of input intervals and output intervals. The input intervals consist of the ranges [pᵢ, pᵢ + dᵢ], and the output intervals consist of the ranges [qᵢ, qᵢ + dᵢ], where the input interval starting with pᵢ and the output interval starting with qᵢ have the same size dᵢ. It describes the bijective function fᵢ that maps a position in an input interval to the position with the same offset in its corresponding output interval by fᵢ(i) = qᵢ + (i − pᵢ), where i 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 move(i, x) = (i', x') evaluates for a given value i ∈ [1, n] and the index x ∈ [1, k] of the input interval containing i the function fᵢ(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 move(i', x'). See again Figure 1 for an example.

To answer move queries on I, we use a move data structure M, which stores three arrays M[p][1..k] = [p₁, p₂, ..., pₖ], M[q][1..k] = [q₁, q₂, ..., qₖ] and M[dx][1..k], where M[dx] stores at position j the index of the input interval containing qᵢ, i.e., M[dx][j] = i iff qᵢ ∈ [pᵢ, pᵢ + dᵢ]. This allows us to compute the i'-part of the query move(M, i, x) = (i', x') by i' = M[q][x] + (i − M[p][x]). To find x', we set x' ← M[dx][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 O(k) steps (see Figure 2). In the following, we show how to reduce the time to answer a move query to 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 \(d = p_{i+a} - q_i\) 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 \(A\) a balancing algorithm, if \(A\) 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 \(A\) be a balancing algorithm and let \(B_a(I)\) be the disjoint interval sequence resulting from an execution of \(A\) on \(I\). Then \(f_{B_a(I)} = f_I\) and \(|B_a(I)| \leq \frac{a}{a-1}k\).

Proof. Let \(t'\) be the number of \(a\)-balancing steps \(A\) 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 \(A\) 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 \(A\) splits \([p_j, p_j + d_j]\) and \([q_j, q_j + d_j]\) at the same offset \(d = p_{i+a} - q_i\) to obtain \(I_t\) from \(I_{t-1}\), clearly \(|I_t| = |I_{t-1}| + 1\) and \(f_I = f_{I_{t-1}}\) and therefore \(|I_t| = k + t\) and \(f_I = f_{I_0} = f_{I_1} = ... = f_{I_t} = f_{B_a(I)}\).

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 \(|B_a(I)| = k + t' \leq \frac{a}{a-1}k\).

Recall that the dominating part of answering a query \(move(i, x) = (i', x')\) 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 \(B_a(I)\), 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 \(O(\frac{a}{a-1}k)\) that allows us to consecutively evaluate \(f_I\) in \(O(a)\) 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 \(T_{in}\) and \(T_{out}\) 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 \(T_{in}\) or \(T_{out}\) we can compute the length of the input- or output-interval it creates by accessing the next pair in \(T_{in}\) or \(T_{out}\), respectively.

To speed up the construction of \(T_{out}\), 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 \(T_{out}\) in \(\mathcal{O}(k)\) additional time by inserting the pairs in the order \((p_{\pi[1]}, q_{\pi[1]}), (p_{\pi[2]}, q_{\pi[2]}), \ldots, (p_{\pi[k]}, q_{\pi[k]})\) into \(T_{out}\).

Let \(T\) be a balanced search-tree over a set \(X\). Then \(\langle v \rangle\) denotes the node in \(T\) storing \(v \in X\). \(T\).has-next(\(\langle v \rangle\)) returns whether \(\langle v \rangle\) has a successor node in \(T\), \(T\).next(\(\langle v \rangle\)) returns this successor node, \(T\).next(\(\langle v \rangle, n\)) returns the \(n\)-th successor of \(\langle v \rangle\) in \(T\), \(T\).pred(\(\langle v \rangle\)) = \(\langle \max\{w \in T \mid w \leq v\}\rangle\), \(T\).succ(\(\langle v \rangle\)) = \(\langle \min\{w \in T \mid v \leq w\}\rangle\), \(T\).min() = \(\langle \min\{v \in T\}\rangle\) and \(\langle p, q\rangle_{in}\) and \(\langle p, q\rangle_{out}\) denote the nodes in \(T_{in}\) and \(T_{out}\) storing \((p, q)\), respectively. The algorithm is \(a\)-heavy expects an output interval \([q_j, q_j + d_j]\) and \(\langle p_i, q_i\rangle_{in}\), 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 \(T_{in}\) starting with \(\langle p_i, q_i\rangle_{in}\) 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 \textbf{while}-loop (lines 3-13), we consider the pairs \((p_i, q_i)\) in \(T_{in}\) and \((p_j, q_j)\) in \(T_{out}\) 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 \(\langle p_i, q_i\rangle_{in}\) and \(\langle p_j, q_j\rangle_{out}\) with the first pairs in \(T_{in}\) and \(T_{out}\), 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 \((p_i, q_i)_{in}\) and \((p_j, q_j)_{out}\), we want to find the next combination of such pairs, that is we want to find \((p_{i'}, q_{i'})\) in \(T_{in}\) and \((p_{j'}, q_{j'})\) in \(T_{out}\), where \(q_{i'}\) is minimal s.t. \(q_{i'} > q_j\) and there exists a pair \((p_i, q_i)_{in}\) in \(T_{in}\), where \(p_{i'}\) is minimal s.t. \(q_{i'} \leq q_{i'} < q_{j'} + d_{j'}\). To find those, we at first iterate further one step with \(\langle p_j, q_j\rangle_{out}\) in \(T_{out}\) (line 8). We then iterate (possibly zero steps) with \(\langle p_i, q_i\rangle_{in}\) in \(T_{in}\), until \(p_i > q_j\) holds, or we have reached the second to last pair in \(T_{in}\) (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 \(\langle p_i, q_i\rangle_{in}\) and \(\langle p_j, q_j\rangle_{out}\) until we find such pairs or until we reach the second to last pair in \(T_{in}\) or \(T_{out}\), respectively.
Algorithm 1 balance-framework().

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

We want to make sure that each output interval \([g_j, q_j + d_j]\) considered in the main while-loop is \(a\)-balanced. To do this, we check if \([g_j, q_j + d_j]\) is \(a\)-heavy by calling is-a-heavy\([g_j, q_j + d_j], (p_i, q_i)_\text{in}\) in line 4. If it is \(a\)-balanced, we proceed to iterate further over \(\mathcal{T}_\text{in}\) and \(\mathcal{T}_\text{out}\) as described to find the next combination of \((p_i, q_i)_\text{in}\) and \((p_j, q_j)_\text{out}\) (lines 7-13). Else, we call balance-up-to\((p_i, q_i)_\text{in}, (p_j, q_j)_\text{out}, q_j\) in line 5, which performs at least one \(a\)-balancing step. More precisely, it splits \([g_j, q_j + d_j]\) at the offset \(d = p_{i+a} - q_j\) into two new output intervals \([g_j, q_j + d]\) 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 \(I'\) denote the number of balancing steps our balancing algorithm performs.

Algorithm 2 balance-up-to\((p_i, q_i)_\text{in}, (p_j, q_j)_\text{out}, q_u).\)

1. \((p_{i+a}, q_{i+a})_\text{in} \leftarrow \mathcal{T}_\text{in}.\text{next}(p_i, q_i)_\text{in}, a;\)
2. \(d \leftarrow p_{i+a} - q_j;\)
3. \(\mathcal{T}_\text{in}.\text{insert}((p_j + d, q_j + d));\)
4. \(\mathcal{T}_\text{out}.\text{insert}(p_j + d, q_j + d);\)
5. if \(p_j + d < q_u\) then
   6. \((p_y, q_y)_\text{out} \leftarrow \mathcal{T}_\text{out}.\text{pred}(p_j + d);\)
   7. \((p_z, q_z)_\text{in} \leftarrow \mathcal{T}_\text{in}.\text{suc}(q_y);\)
   8. if is-a-heavy\((y_j, q_y + d_y), (p_z, q_z)_\text{in}\) then
      9. balance-up-to\((p_z, q_z)_\text{in}, (p_y, q_y)_\text{out}, q_u;\)
1:8 Move-r: Optimizing the r-index

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 $T_{in}$ and $T_{out}$ store $I_s$. Then calling balance-up-to($\langle p_i, q_i \rangle_{in}, \langle p_j, q_j \rangle_{out}, q_u$) updates $T_{in}$ and $T_{out}$ to store $I_t$.

Proof. We at first update $T_{in}$ and $T_{out}$ to store $I_{s+1}$ by finding $\langle p_{i+a}, q_{i+a} \rangle_{in}$ and inserting $(p_j + d, q_j + d)$ into $T_{in}$ and $T_{out}$, 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 $\langle p_y, q_y \rangle_{out}$ with a predecessor search over $T_{out}$ and $\langle p_z, q_z \rangle_{in}$ using a successor search over $T_{in}$. Then, we check if $O_y$ is $a$-heavy in $I_{s+1}$ by calling is-a-heavy($\langle q_y, q_y + d_y \rangle, \langle p_z, q_z \rangle_{in}$). 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($\langle p_z, q_z \rangle_{in}, \langle p_y, q_y \rangle_{out}, q_u$) 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($\langle p_{i'}, q_{i'} \rangle_{in}, \langle p_{j'}, q_{j'} \rangle_{out}, q_u$) 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 $d' = p_{j'} - q_{j'}$. In this call, we update $T_{in}$ and $T_{out}$ 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 $T_{in}$ and $T_{out}$ store $I_s$. Then calling balance-up-to($\langle p_i, q_i \rangle_{in}, \langle p_j, q_j \rangle_{out}, q_j$) updates $T_{in}$ and $T_{out}$ 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 $T_{in}$ and $T_{out}$ (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\((p_z, q_z)_{in}, (p_y, q_y)_{out}, q_j \) in line 12, which updates \( T_m \) and \( T_{out} \) to store \( I_t \) by Lemma 8.

4.1.3 Running time analysis

**Theorem 10.** Given \( T_m \) and \( T_{out} \) for a disjoint interval sequence \( I \) of size \( k \) and an integer \( a \geq 2 \), we can compute \( B_a(I) \) in \( O(k + \frac{k}{a} \log k) \) time and \( O(1) \) additional space.

**Proof.** As argued in Section 3.2, \( T_m \) contains such an \( a \)-balanced disjoint interval sequence \( B_a(I) \) after performing Algorithm 1. Here, we iterate once over \( T_m \) and \( T_{out} \) in total. This takes \( O(k) \) 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 \( \frac{k}{a-1} \) balancing steps. In Algorithm 1, we need \( O(1) \) additional space.

In each of those calls of balance-up-to, we find \( (p_{i+a}, q_{i+a}) \in (O(a) \text{ time}) \), insert \( (p_j + d, q_j + d) \) into \( T_m \) and \( T_{out} \) \( O(\log k) \) time and possibly perform a predecessor- and a successor search over \( T_{out} \) and \( T_m \) \( O(\log k) \) time, respectively. Overall, one call of balance-up-to needs \( O(a + \log k) \) time. We do not need the local variables in balance-up-to, hence balance-up-to needs \( O(1) \) additional space. In total, this yields \( O(k + \frac{k}{a}(a + \log k)) = O(k + \frac{k}{a} \log k) \) time and \( O(1) \) 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 \( M^{LF} \) of size \( r' \geq r \) be a move data structure (Corollary 7) built from the disjoint interval sequence \( B_a(I_{LF}) \), where \( I_{LF} = (l_1, LF(l_1)), (l_2, LF(l_2)), ..., (l_r, LF(l_r)) \). Let \( M^P \) of size \( r'' \geq r \) be a move data structure built from the disjoint interval sequence \( B_a(I_P) \), where \( I_P = (u_1, \Phi(u_1)), (u_2, \Phi(u_2)), ..., (u_r, \Phi(u_r)) \). Let \( L'[1..r'] \) be a string containing the characters \( L[M^{LF}[1]], L[M^{LF}[2]], ..., L[M^{LF}[r']] \).

Note that \( r' \) and \( r'' \) depend on the balancing parameter \( a \). The total used space of these data structures is \( O(r) \), because \( r', r'' \leq \frac{a}{a-1}r = O(r) \) 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 \( b'_i = LF^{-1}(b_i) \), \( e'_i = LF^{-1}(e_i) \), \( \hat{b}_i, \hat{e}_i, \hat{b}'_i \) and \( \hat{e}'_i \) be the indices of the input intervals in \( M^{LF} \) containing \( b_i, e_i, b'_i \) and \( e'_i \), respectively. Define \( z(i) = SA[e_i] \) \( \text{the lexicographically largest suffix of } T \text{ starting with } P_i \), let \( s(i, j) = z(j) + i - j \) and let \( y(i) = \max\{j \in [1, m] | T_{z(i,j)} \text{ starts with } P_i \} \). Finally, define \( b_{m+1}, b_{m+1}, e_{m+1} \) and \( \hat{e}_{m+1} \) analogously, \( i.e., b_{m+1} = 1, b_{m+1} = 1, e_{m+1} = n \) and \( \hat{e}_{m+1} = r' \).

Let \( S \) be a string that occurs in \( T \) and let \( T_i \) be the lexicographically largest suffix of \( T \) among those starting with \( S \). Then we call \( T[1, l + |S|] \) 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 \( s(i, y(i)) = z(i) \).
Move-r: Optimizing the r-index

![Figure 4](image)

**Proof.** Suppose there was a suffix $T_w > T_{y(i+1)}$ starting with $P_i$. Then $T_{w+y(i+1)}$ starts with $P_{y(i)}$ and $T_{w+y(i+1)} = T_{z(y(i+1))}$, contradicting the definition of $T_{z(y(i+1))}$.

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

**Proof.** See Appendix A.

The query `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 $e'_{y(i+1)}$ for $i < m$). If we augment $L'$ with an $O(\log \log \omega)$ time and $O(\omega')$ space `rank` data structure and an $O(1)$ time and $O(\omega')$ space `select` data structure, we can compute $b_i$, $\hat{b}_i$, $e_i$, $\hat{e}_i$, $y(i)$ and $e'_{y(i)}$ in $O(a + \log \log \omega)$ 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 $T_{SA[b_i]}$ 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 $b'_i = \text{select}(L', c, \text{rank}(L', c, b_{i+1} - 1) + 1)$, $b'_i = M_p[b'_i]$ and $(b_i, \hat{b}_i) = \text{move}(M_{\text{LF}}, b'_i, \hat{b}'_i)$. Analogously, we obtain $e_i$ and $\hat{e}_i$ from $e_{i+1}$ and $\hat{e}_{i+1}$ by $e'_i = \text{select}(L', c, \text{rank}(L', c, \hat{e}_{i+1}) + 1) + 1$ and $(e_i, \hat{e}_i) = \text{move}(M_{\text{LF}}, e'_i, \hat{e}'_i)$. 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 $e'_{y(i)}$. If $i = m$, then $y(i) = m$, because $P_m$ occurs at $z(m) \overset{\text{Def.}}{=} s(m, m)$, and therefore $e'_{y(i)} = e'_m$. Else, we have $i < m$. Here, we consider two cases (see Figure 4).

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

In total, we perform two move queries on $\mathcal{M}^\Phi$ ($\mathcal{O}(a)$ 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 if $L'[\hat{b}_{i+1}] = c$ holds, then $b_i' = b_{i+1}$ and $\hat{b}_i' = \hat{b}_{i+1}$. Similarly, $L'[\hat{e}_{i+1}] = c ⇔ e_i' = e_{i+1} ∧ e_i' = \hat{e}_{i+1}$. 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 $\mathcal{O}(a + \log \log \omega \sigma)$ 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 $e_{y(1)}'$ to answer a count query. Those values are only necessary for answering a locate query, which we will discuss next.

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

$$SA_{\Phi}[x_i] = \begin{cases} x_i & p_i \text{ is the end position of a BWT run,} \\ \bot & \text{else.} \end{cases}$$

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

**Proof.** As in Theorem 16, we compute $b_1, e_1, \hat{b}_1$ and $\hat{e}_1$ and additionally $y(1)$ and $e_{y(1)}'$ 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 $\mathcal{O}(a + \log \log \omega \sigma)$ time. Then, we compute

$$s = SA[e_1] \overset{Lem.}{=} 13 s(1, y(1)) \overset{Def.}{=} 12 z(y(1)) + 1 − y(1) \overset{Def.}{=} 12 SA[e_{y(1)}] + 1 − y(1) \overset{Def.}{=} 12 SA[e_{y(1)}]' + y(1) \overset{Def.}{=} 12 [\mathcal{M}^\Phi_p[e_{y(1)}]' + 1] − 1 \overset{Def.}{=} [\mathcal{M}^\Phi_p[SA[e_{y(1)}]']] − y(1)$$

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

Else, we compute the remaining occurrences $[SA[e_{1}−1], SA[e_{1}−2], ..., SA[b_1]] = 1\Phi^c(s), 2\Phi^c(s), ..., m\Phi^c(s)]$ by consecutively performing $e_1 − b_1$ move queries on $\mathcal{M}^\Phi$ starting with $(s, \tilde{s})$ and reporting $s$ after each move query, where $\tilde{s}$ is the index of the input interval in $\mathcal{M}^\Phi$ containing $s$. This takes $\mathcal{O}(\omega c \cdot a)$ additional time. To compute $\tilde{s}$, we at first compute the index $\hat{x} = \mathcal{M}^\Phi_\Phi[SA[e_{y(1)}]']$ of the input interval in $\mathcal{M}^\Phi$ containing $SA[e_{y(1)}]'$. Again, we can
Move-r: Optimizing the r-index

We implemented the algorithms described in Section 4 in C++20. 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 $O(n)$ to $O(|PFP|)$ words, where |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 $\alpha$-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 $occ$ is the average number of occurrences per pattern.

<table>
<thead>
<tr>
<th>text</th>
<th>size [GB]</th>
<th>$\sigma$</th>
<th>$n/r$</th>
<th>$r'/r$</th>
<th>$r''/r$</th>
<th>$N$</th>
<th>$m$</th>
<th>$occ$</th>
</tr>
</thead>
<tbody>
<tr>
<td>einstein.en.txt</td>
<td>0.47</td>
<td>139</td>
<td>1611.18</td>
<td>1.23</td>
<td>1.49</td>
<td>100000</td>
<td>800</td>
<td>748.82</td>
</tr>
<tr>
<td>sars2</td>
<td>84.19</td>
<td>80</td>
<td>686.57</td>
<td>1.38</td>
<td>1.06</td>
<td>10000</td>
<td>50</td>
<td>524838.82</td>
</tr>
<tr>
<td>dewiki</td>
<td>68.72</td>
<td>210</td>
<td>345.80</td>
<td>1.23</td>
<td>1.35</td>
<td>20000</td>
<td>320</td>
<td>313.49</td>
</tr>
<tr>
<td>chr19</td>
<td>58.57</td>
<td>52</td>
<td>272.20</td>
<td>1.06</td>
<td>2.00</td>
<td>150</td>
<td>10</td>
<td>216783.43</td>
</tr>
<tr>
<td>english</td>
<td>2.21</td>
<td>239</td>
<td>3.36</td>
<td>1.19</td>
<td>1.20</td>
<td>30000</td>
<td>35</td>
<td>34.60</td>
</tr>
</tbody>
</table>

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 $I_L$. 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. $\text{rank}(L,c,i)$ is computed by looking up the number of occurrences of $c$ before the $[i/b]$-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 $O(\log l + \log \sigma)$ 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 $\text{rank}(L,c,i)$, 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 so that \( \overline{oc} \approx m \). This implies that when locating those patterns, we measure a blend of LF-, \( \Phi \)- and rank-select queries on \( L' \), since we perform \( \overline{oc} \approx m \) \( \Phi \) queries, \( 2m \) LF queries and at most \( 2m \) rank-select queries on \( L' \). The patterns in the second set were chosen so that \( \overline{oc} \approx 10^5 m \). When locating those patterns, we practically only measure \( \Phi \) 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

A Proof of Lemma 14

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

Proof. Let $p' = SA^{-1}[z(y(i) + 1)]$. By the definition of $P_y(i)$, $T_{z(y(i)) + 1}$ does not start with $P_y(i)$, hence $L[p'] = T[z(y(i)) + 1] \neq P[y(i)]$. 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 $p + 1 < p'$, $T_{SA[p+1]}$ starts with

Move-r: Optimizing the r-index

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

B More practical optimizations

B.1 Practically optimized rank-select data structure

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

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

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

We instead store $\sigma$ 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 $\text{rank}(L', c, i)$ by computing $RS_{L'}[c].\text{rank}(i)$. Similarly, we compute $RS_{L'}[c].\text{select}(i)$ to answer $\text{select}(L', c, i)$.

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 $O(1)$ time and $o(r')$ space $\text{rank}_{L'}$ [23] and $\text{select}_{1}$ [7] data structures. Else, we use an sarray [20] to implement $RS_{L'}[c][1..r']$. This results in $O(\log \frac{\log \sigma}{m} + \log^4 n_c / \log r')$ time to answer a rank query on $L'$ and $O(\log^4 n_c / \log r')$ 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 $O(\sum_{i=1}^m (a + \log \frac{\log \sigma}{m} + \log^4 n_{P[i]} / \log r')) = O(m(a + \log^3 r))$ time to answer a count query, because $1 \leq n_c < r' \forall c \in \Sigma$ and $r' = O(r)$, where $n_{P[i]}$ denotes the number of occurrences of $P[i]$ in $L'$. This is asymptotically worse than the $O(m(a + \log \log_o \sigma))$ 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 $\text{occ} \lesssim m$) 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 $\sum_{c \in \Sigma} (n_c \log (\frac{\log \sigma}{m}) + 2n_c + o(n_c)) = \sum_{c \in \Sigma} (n_c \log (\frac{L'}{n_c})) + 2r' + o(r') = r' \text{H}_0(L') + 2r' + o(r')$ bits, which is asymptotically not larger than a huffman-shaped wavelet-tree of $L'$, which needs $r' \text{H}_0(L') + o(r' \text{H}_0(L') + 1) + \sigma \omega$ bits (see [16]), where $\text{H}_0(L')$ 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 $M_{idx}$ in $O(k' \log k')$ time using a binary search over $M_{p}[1..k']$ for each entry in $M_{idx}[1..k']$. To speed up the construction in practice, we compute $\pi$ for $B_c(I)$ and use it to iterate over the output intervals of $B_c(I)$ in $O(k')$ 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 $(j\text{-th})$ output interval. Hence, we can write $M_{idx}[\pi[j]] \leftarrow i$ for each $j\text{-th}$ output interval. Since the construction of $\pi$ takes $O(k' \log k')$ time, this method still results in the same theoretical time to construct $M_{idx}$, but is much faster in practice.
We can apply a similar method to build $S\Phi$. We at first build the array $S\Phi_{i}[1...r']$, where $S\Phi_{i}[i] = S\Phi[M_{p}(i+1) - 1]$, if $p_{i}$ is the end position of a BWT run and $S\Phi_{i}[i] = \infty$, else. Then, we build the permutation $\pi'[1..r']$ storing the order of the values in $S\Phi_{i}$ in $O(r' \log r')$ time.

To iterate over the output intervals of $M^{\Phi}$, we can reuse $\pi_{\Phi}$ from the construction of $M^{\Phi}_{idx}$, i.e., we can build $S\Phi$ by simultaneously iterating over $M^{\Phi}_{q}[\pi_{\Phi}[1]], M^{\Phi}_{q}[\pi_{\Phi}[2]], ..., M^{\Phi}_{q}[\pi_{\Phi}[r']]$ and $S\Phi_{i}[\pi'[1]], S\Phi_{i}[\pi'[2]], ..., S\Phi_{i}[\pi'[r]]$ in $O(r' + r'') = O(r)$ time and setting $S\Phi_{i}[\pi'[i]] \leftarrow \perp$ for $i \in (r,r']$ in $O(r' - r) = O(r)$ 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 $M_{q}$, we only store the offset of each output interval starting position in the input interval containing it, i.e., we store the arrays $M_{p}[1..k], M_{offs}[1..k]$ and $M_{idx}[1..k]$, where $M_{offs}[i] = q_{i} - M_{p}[M_{idx}[i]]$, for $i \in [1,k]$. This allows us to compute $q_{i}$ with one random access to $M_{p}$ at the position $M_{idx}[i]$, which is irrelevant when evaluating a move query, because when computing $\text{move}(M,i,x) = (i',x')$, we start scanning over $M_{p}$ at the position $M_{idx}[x]$ to find $x'$.

The length of the longest input interval is an upper bound of any value in $M_{offs}$. This implies that we can store $M_{offs}$ with the word width $\omega_{offs}$ by limiting the length of each input interval to $2^{n_{offs}}$. 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 $2^{n_{offs}}$ if it is longer than $2^{n_{offs}}$. 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 $\leq \frac{n}{2^{n_{offs}}}$ splits, the number of input- and output intervals increases by a factor $\leq 1 + \frac{n}{2^{n_{offs}}}$. We choose $\omega_{offs} = \min\{\omega \in \{8,16,24,32,40\} : \frac{n}{\omega} \leq \epsilon\}$ to bound the overall size of the resulting move data structure to $O((1+\epsilon)\frac{n}{\omega-1}k)$, for some $\epsilon > 0$. This suffices, if we assume $n \leq 2^{30}$, that is if the text file is smaller than $\approx 1$TB. In practice, $\epsilon = \frac{1}{8}$ turns out to be reasonable trade-off.

We also store $M_{p}$ and $M_{idx}$ with the minimum possible word widths $\omega_{p} = \min\{\omega \in \{8,16,24,32,40\} : n \leq 2^{\omega}\}$ and $\omega_{idx} = \min\{\omega \in \{8,16,24,32,40\} : k' \leq 2^{\omega}\}$, respectively. Finally, we store $M_{p}, M_{offs}$ and $M_{idx}$ 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 $M^{LF}$. Since $S\Phi_{i}[i] \in [1,r'']$, we can store $S\Phi_{i}$ with the same word width $\omega_{idx}$ as $M^{LF}_{idx}$. 