Non-overlapping Indexing in BWT-Runs Bounded Space

Daniel Gibney[1], Paul Macnichol[2], and Sharma V. Thankachan[2(✉)]

[1] Department of CS, University of Texas at Dallas, Dallas, TX, USA
daniel.gibney@utdallas.edu
[2] Department of CS, North Carolina State University, Raleigh, NC, USA
{pemacnic, svalliy}@ncsu.edu

Abstract. We revisit the non-overlapping indexing problem for an efficient repetition-aware solution. The problem is to index a text $T[1..n]$, such that whenever a pattern $P[1..p]$ comes as a query, we can report the largest set of non-overlapping occurrences of P in T. A previous index by Cohen and Porat [ISAAC 2009] takes linear space and optimal $O(p + \mathsf{occ_{no}})$ query time, where $\mathsf{occ_{no}}$ denotes the output size. We present an index of size $O(r)$, where r denotes the number of runs in the Burrows Wheeler Transform (BWT) of T. The parameter r is significantly smaller than n for highly repetitive texts. The query time of our index is $O(p \log \log_w \sigma + \mathsf{sort}(\mathsf{occ_{no}}))$, where σ denotes the alphabet size, w denotes the machine word size in bits and $\mathsf{sort}(x)$ denotes the time for sorting x integers within the range $[1, n]$.

1 Introduction and Related Work

Text indexing is a well-studied problem in computer science with many applications in information retrieval and bioinformatics. The basic version is defined as follows: Preprocess a given text $T[1..n]$ into a data structure (called index) such that whenever a pattern $P[1..p]$ comes as an input, we can efficiently support both *counting* queries and *reporting* queries. A reporting query asks to output $Occ(T, P) = \{i \mid T[i..i+p) = P\}$, the set of occurrences of P in T and a counting query asks for its size *occ*. We assume that the characters in T and P are from an alphabet $\Sigma = \{0, 1, 2, \ldots, \sigma - 1\}$ and $\sigma = n^{O(1)}$. Our model of computation is word RAM with a machine word of size $w = \Omega(\log n)$ bits.

By maintaining the classic suffix tree data structure over T, we can perform both counting and reporting in optimal times $O(p)$ and $O(p + occ)$, respectively [25]. Alternatively, we can use the suffix array of T for counting in time $O(p \log n)$ and reporting in time $O(p \log n + occ)$ [19]. The space complexity of both structures is $O(n)$ words, equivalently $O(n \log n)$ bits, which can be orders of magnitude more than the size of text, which is $n\lceil \log \sigma \rceil$ bits. Therefore, obtaining space-efficient encoding of these fundamental data structures has been an active line of research. Two important results on this topic from early 2000 are the Compressed Suffix Arrays and the FM index—encodings in succinct

F. M. Nardini et al. (Eds.): SPIRE 2023, LNCS 14240, pp. 260–270, 2023.
https://doi.org/10.1007/978-3-031-43980-3_21

or entropy-compressed space [7,13]. Compressed suffix tree was also introduced later [24]. These initial results have witnessed various improvements over time; we refer to [20] for further reading. One of the recent breakthroughs in (compressed) text indexing is the *r-index* by Gagie, Navarro, and Prezza [8]. Its $O(r)$ space version can perform counting and reporting in times $O(p \log \log_w(\sigma + n/r))$ and $O((p + occ) \log \log_w(\sigma + n/r))$ respectively, where r denotes the number of runs in the text's Burrows-Wheeler Transform (BWT). The parameter r is a popular measure of compressibility that captures repetitiveness. It can be significantly smaller than n for highly repetitive texts. In a new result by Nishimoto and Tabei [22], the r-index's query times for counting and reporting were improved to $O(p \log \log_w \sigma)$ time and $O(p \log \log_w \sigma + occ)$, respectively. Another result by Gagie et al. [9] shows that the suffix tree can be encoded in $O(r \log(n/r))$ space and support most of its operations in time $O(\log(n/r))$, which includes random access to suffix array, inverse suffix array, longest common prefix array, etc.

We now formally define the main problem considered in this paper.

Problem 1 (Non-overlapping indexing). *Preprocess a given text $T[1..n]$ over an integer alphabet of size $\sigma = n^{O(1)}$ into a data structure (called index) such that whenever a pattern $P[1..p]$ comes as a query, we can report the largest set $Occ_{no}(T, P) \subseteq Occ(T, P)$ of occurrences of P in T, such that the difference between any two occurrences in $Occ_{no}(T, P)$ is at least p.*

Keller et al. [16] introduced this problem and presented an $O(n \log n)$ space solution with $O(p + \mathsf{occ_{no}} \cdot \log \log n)$ query time, where $occ_{no} = |Occ_{no}(T, P)|$. In 2009, Cohen and Porat proposed an improved solution with space $O(n)$ and optimal $O(p + \mathsf{occ_{no}})$ query time [4]. Later, Ganguly et al. [10,11] showed that all we need is a suffix tree (or any of its space-efficient variants) of T. The time complexity of their query algorithm is $O(search(P) + \mathsf{occ_{no}} \cdot \mathsf{t_{SA}} + \mathsf{sort}(\mathsf{occ_{no}}))$, where $search(P)$ denotes the time for computing the suffix range of P and $\mathsf{t_{SA}}$ denotes the time for accessing a given entry in the suffix array or inverse suffix array, and $\mathsf{sort}(x)$ denotes the time for sorting a subset of $\{1, 2, \ldots, n\}$ of size x. Many space-time trade-offs are immediate from this general result, including a repetition-aware index of size $O(r \log(n/r))$ and query time $O(p + \mathsf{occ_{no}} \cdot \log(n/r) + \mathsf{sort}(\mathsf{occ_{no}}))$ using the suffix tree of Gagie et al. [9]. The interesting question is, can we improve the space complexity to $O(r)$? Note that Ganguly et al.'s algorithm [11] needs random access to the suffix array and its inverse array, and whether suffix trees can be encoded in $O(r)$ space is still open. To that end, we present the following result.

Theorem 1. *For the non-overlapping indexing problem, there exists an $O(r)$ space index that can report $Occ_{no}(T, P)$ in time $O(p \log \log_w \sigma + \mathsf{sort}(\mathsf{occ_{no}}))$.*

Our result is based on the work by Hooshmand et al. [15], where the authors proposed modifying Ganguly et al.'s algorithm [11], which led to an efficient external memory solution; also see [14]. This modified algorithm avoids much of the random accesses but requires some additional structures, specifically the suffix array of the reverse of T, for its implementation. The critical insight we

make in this paper is that the (less general) operations supported by the r-index of T suffice to efficiently implement the algorithm by Hooshmand et al. [15].

2 Preliminaries

For a string $S[1..m] \in \Sigma^m$, we denote its i-th character by $S[i]$, and a substring starting at position i and ending at position j by $S[i..j]$, which is an empty string if $i > j$. When $S[i..j]$ is a suffix of S (i.e., $j = m$), we denote it by $S[i..]$ and when $S[i..j]$ is a prefix of S (i.e., $i = 1$), we denote it by $S[..j]$. The reverse of S is denoted by $\overleftarrow{S}$. The concatenation of two strings (or characters) S_1 and S_2 is denoted by S_1S_2.

2.1 Rank and Select

For any string $S[1..m] \in \Sigma^m$, $rank_S(i, c)$ denotes the number of occurrences of c in $S[1..i]$, where $i \in [1, m]$, $c \in \Sigma$. Also, $select_A(j, c)$ denotes the ith occurrence of c in S. A rank query of the form $rank_S(i, S[i])$ is called a partial query.

If S is a binary string with t 1's, we can maintain a $t \log(m/t) + O(t)$-bit structure (known as indexible dictionary) and find $rank_S(i, 1)$ for any i with $S[i] = 1$ in $O(1)$ time [23]. It can also support select queries in $O(1)$ time.

2.2 Suffix Array

The suffix array of a text $T[1..n]$ is an array $SA[1, n]$, such that $SA[i]$ represents the starting position of the ith smallest suffix of T in lexicographic order. For convenience, we assume that the last character of T, denoted by \$, does not appear anywhere else in the text or in the pattern and is lexicographically smaller than all other symbols in Σ. The suffix range of a pattern $P[1..p]$, denoted by $[sp(P), ep(P)]$ is the maximal range, such that $Occ(T, P) = \{SA[i] \mid i \in [sp(P), ep(P)]\}$. The suffix range is empty if P does not appear in T. The suffix range, hence the number of occurrences, can be computed in $O(p \log n)$ time. The inverse suffix array ISA is also an array of length n, such that $ISA[SA[i]] = i$ for all $i \in [1, n]$; equivalently, $ISA[i]$ is the lexicographic rank of the suffix $T[i..]$.

2.3 Burrows–Wheeler Transform

The Burrows-Wheeler Transform (BWT) [3] of a text T is a (reversible) permutation of the symbols of T such that $BWT[i] = T[SA[i] - 1]$ if $SA[i] \neq 1$ and is $T[n]$ otherwise (recall that $T[n] =$ \$ appears only once in T and is smaller than all other symbols in lexicographic order). The BWT can be encoded in $n \log \sigma$ bits or even in $O(r)$ words by applying run-length encoding, where $r \in [\sigma, n]$ denotes the number of runs (maximal unary substrings) in BWT. For example, the BWT of the text $mississippi\$$ is $ipssm\$pissii$ with 9 runs. The LF-mapping is a function defined as follows: $LF[i]$ is $ISA[SA[i] - 1]$ if $SA[i] \neq 1$ and is 1 otherwise. The LF-mapping can be computed using rank

queries on BWT as follows: $LF[i] = Count[BWT[i]] + rank_{BWT}(i, BWT[i])$, where $Count[c] = |\{k \in [1,n] \mid T[k] < c\}|$ for any $c \in \Sigma$. We call $i \in [1,n]$ a run boundary, if $i \in \{1,n\}$ or $BWT[i] \neq BWT[i-1]$ or $BWT[i] \neq BWT[i+1]$.

2.4 The r-Index and Some Related Results

Using the **r-index** by Gagie et al. [8,9] and refinements by Bannai et al. [2], we can support the following operations:

1. Given a pattern $P[1..p]$, for each $j \in [1,p]$, we can compute the suffix range of $P[j..p]$. i.e., $[sp(P[j..]), ep(P[j..])]$, in total time $O(p \log\log_w(\sigma + n/r))$. In addition to this, we can get $SA[sp(P[j..])]$ and $SA[ep(P[j..])]$ for each $j \in [1,p]$ in the same time.
2. Given any i, we can compute $LF[i]$ in $O(\log\log_w(n/r))$ time.
3. Given any $(i, SA[i])$, we can compute $\phi^{-1}(SA[i]) = SA[i+1]$ in $O(\log\log_w(n/r))$ time.

Nishimoto and Tabei [22] improved the time complexity of operation 1 to $O(p \log\log_w \sigma)$, and operations 2 and 3 to $O(1)$ time. As a result, given any $(i, i+h, SA[i])$, we can report $\{SA[k] \mid i \leq k \leq i+h\}$ in $O(h+1)$ time. Since the result for operation 1 is not explicitly stated in their paper, especially $SA[ep(\cdot)]$ part, we provide a short proof here.

Lemma 1 (Modified Toehold Lemma). *By maintaining some additional information with r-index in $O(r)$ space, we can support the following query: given a pattern $P[1..p]$, we can output $SA[sp(P[j..])]$ and $SA[ep(P[j..])]$ for all $j \in [1,p]$ in time $O(p \log\log_w \sigma)$.*

Proof. We store a bit vector $B[1..n]$ and a sampled suffix array SA'. The vector B is defined as follows: $B[LF[i]] = 1$ iff i is a run boundary. Therefore, number of 1's in B is $\Theta(r)$. By maintaining B in space $O(r \log(n/r))$ bits, i.e., $O(r)$ words, we can compute $rank_B(i,1)$ for any i with $B[i] = 1$ in $O(1)$ time (via a partial rank query) [23]. The sampled suffix array SA' is defined as, $SA'[j] = SA[select_B(j,1)]$ and its size is $O(r)$. Therefore, $SA[LF[i]] = SA'[rank_B(LF[i],1)]$ for any run boundary i can be retrieved in $O(1)$ time. We also explicitly store $Count[c]$ for all $c \in \Sigma$.

We process a query $P[1..p]$ as follows. Inductively, assume that we have already computed $sp(P[k..]), ep(P[k..])$, $SA[sp(P[k..])]$ and $SA[ep(P[k..])]$ for all $k \in [j,p]$ for some $j \leq p$ (the base case where $k = p$ is easy). The r-index can give us $[sp(P[k-1..]), ep(P[k-1..])]$ in $O(\log\log_w \sigma)$ time. Let α be the first and β be the last occurrences of $P[k-1]$ in the range $[sp(P[k..]), ep(P[k..])]$ in BWT. Note that since BWT is run-length encoded form; finding α and β is costly, however we have $LF[\alpha] = sp(P[k-1..])$ and $LF[\beta] = ep(P[k-1..])$. Also observe that finding $BWT[x]$ for an arbitrary x is costly. However, we can utilize the $O(1)$ time LF-mapping operation to determine if $\mathsf{BWT}[x]$ equals $P[k-1]$, since $BWT[x] = P[k-1]$ iff $Count(P[k-1]) < LF[x] \leq Count(P[k-1]+1)$. We have the following cases:

- If $BWT[sp(P[k..])] = P[k-1]$, then $SA[sp(P[k-1..])] = SA[sp(P[k..])] - 1$. Else, α will be a run boundary and $SA[sp(P[k-1..])] = SA[LF[\alpha]] = SA'[rank_B(LF[\alpha], 1)]$ can be obtained in constant time.
- If $BWT[ep(P[k..])] = P[k-1]$, then $SA[ep(P[k-1..])] = SA[ep(P[k..])] - 1$. Else, β will be a run boundary and $SA[ep(P[k-1..])] = SA[LF[\beta]] = SA'[rank_B(LF[\beta], 1)]$ can be obtained in constant time.

This completes the proof. □

3 The Data Structures

Let $x_1, x_2, \ldots, x_{occ}$ denotes the occurrences of $P[1..p]$ in T in the ascending order. We say x_i and x_j, where $i < j$ are overlapping occurrences if $0 < x_j - x_i < p$ and non-overlapping occurrences otherwise. Define, $Overlap(x_i, x_j) = \max\{p - (x_j - x_i), 0\}$. The following simple algorithm can report the largest set of non-overlapping occurrences. First, find all occurrences of P and sort them to obtain $x_1, x_2, \ldots, x_{occ}$. Report the last occurrence x_{occ}. Then scan the remaining occurrences in the right-to-left order, and report an occurrence if it does not overlap with the last reported occurrence. Although this algorithm correctly reports Occ_{no}, its time complexity is equal to the time for reporting all occurrences of P plus $\mathsf{sort}(occ)$. For a better solution, we exploit the pattern's periodicity.

The period of $P[1..p]$ is its shortest prefix Q, such that we can write P as a concatenation of several copies of Q and a proper prefix R of Q. Note that R can be an empty string. For example, we can write $P = abcabcab$ as Q^2R, where $Q = abc$ and $R = ab$. Also, define $\lambda = \lceil p/|Q| \rceil$. Also, we say P is *periodic* if $\lambda > 2$ and *aperiodic* otherwise. We can determine P's period in $O(p)$ time [5]. If P is aperiodic, then $occ = \Theta(\mathsf{occ_{no}})$ and the result of Theorem 1 is immediate using r-index and the simple algorithm described before. The rest of this paper focuses only on the more involved periodic case.

If P is periodic and $Overlap(x_{i+1}, x_i) \geq |Q|$, then $x_{i+1} - x_i = |Q|$. Based on this, we have the following definition from [11].

Definition 1 (Cluster). *Let $1 \leq i \leq j \leq occ$ and P is periodic. We call a subset $\{x_i, x_{i+1}, \ldots, x_j\}$ of consecutive occurrences a cluster, iff*

1. *$i = 1$ or $Overlap(x_{i-1}, x_i) < |Q|$,*
2. *$x_{k+1} - x_k = |Q|$ for all $k \in [i, j)$, and*
3. *$j = occ$ or $Overlap(x_j, x_{j+1}) < |Q|$.*

Additionally, we call x_i (resp., x_j) the head *(resp,* tail*) of the cluster.*

We use π to denote the number of clusters. Let $h_1, h_2, \ldots, h_\pi$ denotes the clusters heads and $t_1, t_2, \ldots, t_\pi$ denotes clusters tails, where $h_1 \leq t_1 < h_2 \leq t_2 < \ldots, < h_\pi \leq t_\pi$. Define $C_i = \{h_i, h_i + |Q|, h_i + 2|Q|, \ldots, t_i\}$, which call the ith cluster. Note that two consecutive non-overlapping occurrences within the same cluster must be exactly $\lambda|Q|$ characters apart.

3.1 An $O(r \log(n/r))$ Space Solution

We obtain the following result in this section via a direct implementation of Ganguly et al.'s algorithm [11] using the $O(r \log(n/r))$ space suffix tree of Gagie et al. [9]. The algorithm is based on the following observations:

- The number of clusters $\pi = O(\mathsf{occ_{no}})$; follows from the fact that $\{h_1, h_3, h_5, \dots\}$ is a set of non-overlapping occurrences of size $\lceil \pi/2 \rceil$.
- The set $\{t_1, t_2, \dots, t_\pi\}$ of all cluster tails can be obtained using a suffix tree (or an equivalent data structure) efficiently as described below.
- Once we have sorted the list of all cluster tails, we can find Occ_{no} via $O(\mathsf{occ_{no}})$ number of ISA queries.

We now present the algorithm formally.

1. Find all cluster tails and sort them to obtain $t_1, t_2, \dots, t_\pi$ (also let $t_0 = 0$).
2. Initialize $x = \infty$ (we use this variable to keep track of the last reported occurrence).
3. For $i = \pi$ to 1, process C_i as follows:
   (a) If x and t_i are non-overlapping, then $x = t_i$; otherwise $x = t_i - |Q|$ (this new x is potentially the rightmost output from C_i).
   (b) While $x \in C_i$ (i.e., $ISA[x] \in [sp(P), ep(P)]$ and $t_{i-1} < x$) report x and $x \leftarrow x - |Q|\lambda$.

To find all cluster tails, observe that an occurrence of P is a cluster tail iff it is not an occurrence of QP. Therefore, $\{t_1, t_2, \dots, t_\pi\} = \{SA[k] \mid k \in [sp(P), ep(P)]$ and $k \notin [sp(QP), ep(QP)]\}$. Since P is a prefix of QP, we have $[sp(QP), ep(QP)] \subseteq [sp(P), ep(P)]$. Therefore,

$$\{t_1, t_2, \dots, t_\pi\} = \{SA[k] \mid sp(P) \leq k < sp(QP), ep(QP) < k \leq ep(P)\}.$$

The implementation is straightforward; step-1 takes $O(\pi)$ number of SA queries and the step-3 takes $O(\mathsf{occ_{no}})$ number of SA queries. This combined with the time initial pattern search and the sorting of all cluster tails, the query time can be bounded by $O(p + \mathsf{occ_{no}} \cdot \log(n/r) + \mathsf{sort}(\mathsf{occ_{no}}))$.

3.2 An $O(r + r^R)$ Space Solution

This result is based on a slight "modification" of Ganguly et al.'s algorithm [11], which was proposed by Hooshmand et al. [15] for efficiently solving the non-overlapping indexing problem in the external memory model by minimizing the number of SA/ISA queries. Some key observations on the previous algorithm are as follows:

- Step-1 (of finding all cluster tails) can be implemented using r-index (using ϕ^{-1} queries instead of SA queries).

– Once we have the sorted list of all cluster heads, we can avoid the ISA query in Step-3(b), because $x \in C_i$ iff $h_i \leq x$.

Formally, we have the following algorithm with a slight modification.

1. Find all cluster tails and sort them to obtain $t_1, t_2, \ldots, t_\pi$.
2. Find all cluster heads and sort them to obtain $h_1, h_2, \ldots, h_\pi$.
3. Initialize $x = \infty$.
4. For $i = \pi$ to 1, process C_i as follows:
   (a) If x and t_i are non-overlapping, then $x = t_i$; otherwise $x = t_i - |Q|$ (this new x is potentially the rightmost non-overlapping occurrence from C_i).
   (b) While $x \in C_i$ (i.e., $h_i \leq x$), report x and $x \leftarrow x - |Q|\lambda$.

We now present the implementation details. We execute step-1 using the r-index of T as follows. Find the suffix range $[sp(P), ep(P)]$ of P and the suffix range $[sp(QP), ep(QP)]$ of QP. We also obtain $SA[sp(P)]$ and $SA[ep(QP)]$ (refer to Lemma 1). Then, all cluster tails can be obtained by applying ϕ^{-1} function π times. The time complexity is $O(p \log \log_w \sigma + \pi)$ plus $\mathsf{sort}(\pi)$. For Step-2, we use the following strategy by Hooshmand et al. [15]. An occurrence x of P is a cluster head iff $(x - |Q|)$ is not an occurrence of QP. Alternatively, we can say, $i \in [1, n]$ is a cluster head iff a substring of T ending at $(i + p - 1)$ matches with P, but not QP. The position $(i + p - 1)$ in T corresponds to an occurrence $(n - (i + p - 1) + 1)$ of $\overleftarrow{P}$, but $\overleftarrow{QP}$ in $\overleftarrow{T}$. This means, cluster heads are equivalent to cluster tails in the reverse text, and we can retrieve them using the strategy used before, but on the suffix tree (or r-index) of the reverse text. Therefore, the time complexity is also $O(p \log \log_w \sigma + \pi)$ plus $\mathsf{sort}(\pi)$. Step-4 takes $(\mathsf{occ_{no}})$ time and the overall time is $O(p \log \log_w \sigma + \mathsf{sort}(\mathsf{occ_{no}}))$.

Since we maintain two r-indexes, the space complexity is $O(r + r^R)$, where r^R is the number of runs in the BWT of $\overleftarrow{T}$. Note that r^R can be more than r (see [12]) although a recent result shows that $r^R = O(r \log^2 n)$ [17]. Therefore, the space complexity (in terms of r and n) is $O(r \log^2 n)$.

3.3 Our Final $O(r)$ Space Solution

In this section, we prove that by maintaining an $O(r)$ space structure and the r-index of T, we can find all cluster heads in time $O(p \log \log_w \sigma + \pi)$. Therefore, for implementing Step-2 of the previous algorithm in Sect. 3.2, the r-index for the text's reverse is no longer required; hence Theorem 1 is immediate.

Recall that a position x is a cluster head iff x is an occurrence of P and $x - |Q|$ is not an occurrence of QP. This means, x is a cluster head, iff there exists a proper (possibly empty) suffix $Q[j..]$ of Q (i.e., $j \in [2, |Q| + 1]$), such that $y = (x - (|Q| - j + 1))$ is an occurrence of $Q[j..]P$ and $T[y - 1] \neq Q[j - 1]$. We have the following observation by substituting $SA[i] = y$.

Observation 1. *For some $j \in [2, |Q| + 1]$, $SA[i]$ is an occurrence of $Q[j..]P$ and $BWT[i] \neq Q[j - 1]$ iff $SA[i] + (|Q| - j + 1)$ is a cluster head.*

The set of cluster heads is given by the union of $\Pi_2, \Pi_3, \ldots, \Pi_{|Q|+1}$, where

$$\Pi_j = \{SA[i]+(|Q|-j+1) \mid i \in [sp(Q[j..]P), ep(Q[j..]P)] \text{ and } BWT[i] \neq Q[j-1]\}.$$

Lemma 2 presents our structure for finding Π_j for any j in optimal $O(1+|\Pi_j|)$ time, given $sp(Q[j..]P), ep(Q[j..]P)$ and $SA[sp(Q[j..]P)]$. Finding these input parameters for all values of $j \in [2, |Q|+1]$ using r-index takes $O(p \log \log_w \sigma)$ time. Thus, the overall time for finding all cluster heads is $O(p \log \log_w \sigma + |Q| + \sum_j |\Pi_j|) = O(p \log \log_w \sigma + \pi)$ as desired.

Lemma 2. *By maintaining an $O(r)$ space structure with r-index, we can support the following query: given a range $[sp, ep]$, $SA[sp]$ and a character $c \in \Sigma$, we can output the elements in $X = \{SA[i] \mid i \in [sp, ep]$ and $BWT[i] \neq c\}$ in optimal $O(1 + |X|)$ time.*

Proof. We maintain a sorted list $L[1, r]$ of the start of all run boundaries (i.e., i's, where $i = 1$ or $BWT[i-1] \neq BWT[i]$. We also maintain a sampled suffix array $SA'[1, r]$, where $SA'[i] = SA[L[i]]$. We now present the query algorithm.

If $ep - sp = LF[ep] - LF[sp]$, we conclude that all characters in $BWT[sp, ep]$ are the same. Then, if $BWT[sp] \neq c$, we report $SA[sp]$ and all the remaining entries in $SA[sp, ep]$ using Φ^{-1} function, else, we report none of them. On the other hand, if $ep - sp \neq LF[ep] - LF[sp]$, there exists two values f and h, such that $L[f-1] \leq sp < L[f] \leq L[f+h] \leq ep < L[f+h+1]$. We find f via binary search in time $O(\log r)$ and then find h in $O(h)$ time. Then, perform the steps below.

1. If $BWT[sp] \neq c$, then report $SA[sp]$, compute the remaining entries in $SA[sp, L[f])$ using Φ^{-1} function, and report them.
2. For all $g \in [f, f+h)$, if $BWT[L[g]] \neq c$, then report $SA[L[g]] = SA'[g]$, compute the remaining entries in $SA[L[g], L[g+1])$ using Φ^{-1} function, and report them.
3. If $BWT[L[f+h]] \neq c$, then report $SA[L[f+h]] = SA'[f+h]$, compute the remaining entries in $SA[L[g], ep]$ using Φ^{-1} function, and report them.

The time complexity is $O(\log r + h + |X|)$. Also note that for any g, $BWT[L[g]] \neq BWT[L[g+1]]$. Therefore, $|X| \geq (h-1)/2$.

Finally, to remove the term $\log r$, we maintain some additional structures: (i) the optimal one-dimensional range reporting structure by Alstrup et al. [1] over L in $O(r)$ space and (ii) a bit vector $B[1..n]$, such that $B[j] = 1$ iff $j = L[i]$ for some $i \in [1, r]$. We maintain B in space $O(r \log(n/r))$ bits, i.e., $O(r)$ words, so that partial rank queries ($rank_B(j, 1)$ when $B[j] = 1$) can be computed in $O(1)$ time [23]. Now, for computing f and h, we use the following procedure: report all $L[i]$'s within $(sp, ep]$ in time $O(h)$. The smallest among them is $L[f]$ and the largest among them is $L[f+h]$. Then compute $f = rank_B(L[f], 1)$ and $f + h = rank_B(L[f+h], 1)$ using two partial rank queries. The overall time complexity is optimal as desired. □

4 Open Problems

We conclude with some follow-up questions for future research.

1. Can we design an efficient index for *counting* the largest number of non-overlapping occurrences of P in T? i.e., an index that can quickly output $\mathsf{occ}_{\mathsf{no}}$. No nontrivial result is known for this problem; therefore, it is interesting to know whether there exists an $O(n \cdot poly \log(n))$ space index with query time $O(p \cdot poly \log(n))$.
2. Can we design new space-time trade-offs for the non-overlapping indexing problem, where space is in terms of other measures of repetitiveness, like the number of Lempel-Ziv factors [26] or δ-measure [18] (a.k.a. substring complexity)?
3. Can we design repetition-aware indexes for the *range non-overlapping indexing* problem, which is a generalization of the non-overlapping indexing problem? Here the input consists of a pattern P and a range $[\alpha, \beta]$, and the task is to output the largest set of non-overlapping occurrences within the range $[\alpha, \beta]$. Several solutions exist to this problem [4,6,16], including an $O(n \log^{\epsilon} n)$ space index with optimal query time [11] and a linear-space index with near-optimal query time [21], where $\epsilon > 0$ denotes an arbitrarily small constant. An orthogonal range query data structure is a part of these indexes, which makes it challenging to encode them in repetition-aware space.

Acknowledgements. This research is supported in part by the U.S. National Science Foundation (NSF) award CCF-2315822.

References

1. Alstrup, S., Brodal, G.S., Rauhe, T.: Optimal static range reporting in one dimension. In: Proceedings on 33rd Annual ACM Symposium on Theory of Computing, 6–8 July 2001, Heraklion, Crete, Greece, pp. 476–482 (2001). http://doi.acm.org/10.1145/380752.380842, https://doi.org/10.1145/380752.380842
2. Bannai, H., Gagie, T., Tomohiro, I.: Refining the r-index. Theor. Comput. Sci. **812**, 96–108 (2020). https://doi.org/10.1016/j.tcs.2019.08.005
3. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. SRC Research Report, 124 (1994)
4. Cohen, H., Porat, E.: Range non-overlapping indexing. In: Proceedings of the Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, 16–18 December 2009, pp. 1044–1053 (2009). http://dx.doi.org/10.1007/978-3-642-10631-6_105, https://doi.org/10.1007/978-3-642-10631-6_105
5. Crochemore, M.: String-matching on ordered alphabets. Theoret. Comput. Sci. **92**(1), 33–47 (1992)
6. Crochemore, M., Iliopoulos, C.S., Kubica, M., Rahman, M.S., Walen, T.: Improved algorithms for the range next value problem and applications. In: Proceedings of the STACS 2008, 25th Annual Symposium on Theoretical Aspects

of Computer Science, Bordeaux, France, 21–23 February 2008, pp. 205–216 (2008). http://dx.doi.org/10.4230/LIPIcs.STACS.2008.1359, https://doi.org/10.4230/LIPIcs.STACS.2008.1359
7. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005). http://doi.acm.org/10.1145/1082036.1082039, https://doi.org/10.1145/1082036.1082039
8. Gagie, T., Navarro, G., Prezza, N.: Optimal-time text indexing in BWT-runs bounded space. In: Czumaj, A. (ed.) Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, 7–10 January 2018, pp. 1459–1477. SIAM (2018). https://doi.org/10.1137/1.9781611975031.96
9. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in BWT-runs bounded space. J. ACM **67**(1), 2:1–2:54 (2020). https://doi.org/10.1145/3375890
10. Ganguly, A., Shah, R., Thankachan, S.V.: Succinct non-overlapping indexing. In: Cicalese, F., Porat, E., Vaccaro, U. (eds.) CPM 2015. LNCS, vol. 9133, pp. 185–195. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19929-0_16
11. Ganguly, A., Shah, R., Thankachan, S.V.: Succinct non-overlapping indexing. Algorithmica **82**(1), 107–117 (2020). https://doi.org/10.1007/s00453-019-00605-5
12. Giuliani, S., Inenaga, S., Lipták, Z., Prezza, N., Sciortino, M., Toffanello, A.: Novel results on the number of runs of the burrows-wheeler-transform. In: Bureš, T., et al. (eds.) SOFSEM 2021. LNCS, vol. 12607, pp. 249–262. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67731-2_18
13. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2), 378–407 (2005). https://doi.org/10.1137/S0097539702402354
14. Hooshmand, S., Abedin, P., Külekci, M.O., Thankachan, S.V.: Non-overlapping indexing - cache obliviously. In: Navarro, G., Sankoff, D., Zhu, B. (eds.) Annual Symposium on Combinatorial Pattern Matching, CPM 2018, 2–4 July 2018 - Qingdao, China. LIPIcs, vol. 105, pp. 8:1–8:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.CPM.2018.8
15. Hooshmand, S., Abedin, P., Külekci, M.O., Thankachan, S.V.: I/O-efficient data structures for non-overlapping indexing. Theor. Comput. Sci. **857**, 1–7 (2021). https://doi.org/10.1016/j.tcs.2020.12.006
16. Keller, O., Kopelowitz, T., Lewenstein, M.: Range non-overlapping indexing and successive list indexing. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 625–636. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73951-7_54
17. Kempa, D., Kociumaka, T.: Resolution of the burrows-wheeler transform conjecture. In: Irani, S. (ed.) 61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, 16–19 November 2020, pp. 1002–1013. IEEE (2020). https://doi.org/10.1109/FOCS46700.2020.00097
18. Kociumaka, T., Navarro, G., Prezza, N.: Toward a definitive compressibility measure for repetitive sequences. IEEE Trans. Inf. Theory **69**(4), 2074–2092 (2023). https://doi.org/10.1109/TIT.2022.3224382
19. Manber, U., Myers, E.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993). https://doi.org/10.1137/0222058
20. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. **39**(1), 2 (2007). https://doi.org/10.1145/1216370.1216372

21. Nekrich, Y., Navarro, G.: Sorted range reporting. In: Fomin, F.V., Kaski, P. (eds.) SWAT 2012. LNCS, vol. 7357, pp. 271–282. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31155-0_24
22. Nishimoto, T., Tabei, Y.: Optimal-time queries on BWT-runs compressed indexes. In: Bansal, N., Merelli, E., Worrell, J. (eds.) 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, 12–16 July 2021, Glasgow, Scotland (Virtual Conference). LIPIcs, vol. 198, pp. 101:1–101:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ICALP.2021.101
23. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms **3**(4), 43 (2007). https://doi.org/10.1145/1290672.1290680
24. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst. **41**(4), 589–607 (2007). https://doi.org/10.1007/s00224-006-1198-x
25. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, 15–17 October 1973, pp. 1–11 (1973). http://dx.doi.org/10.1109/SWAT.1973.13, https://doi.org/10.1109/SWAT.1973.13
26. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)