FM-Index Reveals the Reverse Suffix Array

Arnab Ganguly
Department of Computer Science, University of Wisconsin - Whitewater, WI, USA
gangulya@uww.edu

Daniel Gibney
Department of Computer Science, University of Central Florida, Orlando, FL, USA
Daniel.Gibney@ucf.edu

Sahar Hooshmand
Department of Computer Science, University of Central Florida, Orlando, FL, USA
sahar@cs.ucf.edu

M. Oğuzhan Külekci
Informatics Institute, Istanbul Technical University, Turkey
kulekci@itu.edu.tr

Sharma V. Thankachan
Department of Computer Science, University of Central Florida, Orlando, FL, USA
sharma.thankachan@ucf.edu

Abstract

Given a text $T[1, n]$ over an alphabet Σ of size σ, the suffix array of T stores the lexicographic order of the suffixes of T. The suffix array needs $\Theta(n \log n)$ bits of space compared to the $n \log \sigma$ bits needed to store T itself. A major breakthrough [FM–Index, FOCS'00] in the last two decades has been encoding the suffix array in near-optimal number of bits ($\approx \log \sigma$ bits per character). One can decode a suffix array value using the FM-Index in $\log^{O(1)} n$ time.

We study an extension of the problem in which we have to also decode the suffix array values of the reverse text. This problem has numerous applications such as in approximate pattern matching [Lam et al., BIBM' 09]. Known approaches maintain the FM–Index of both the forward and the reverse text which drives up the space occupancy to $2n \log \sigma$ bits (plus lower order terms). This brings in the natural question of whether we can decode the suffix array values of both the forward and the reverse text, but by using $n \log \sigma$ bits (plus lower order terms). We answer this question positively, and show that given the FM–Index of the forward text, we can decode the suffix array value of the reverse text in near logarithmic average time. Additionally, our experimental results are competitive when compared to the standard approach of maintaining the FM–Index for both the forward and the reverse text. We believe that applications that require both the forward and reverse text will benefit from our approach.

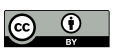
2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Data Structures, Suffix Trees, String Algorithms, Compression, Burrows–Wheeler transform, FM-Index

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.13

Supplementary Material `https://github.com/oguzhankulekci/reverseSA`

Funding This research is supported in part by the U.S. National Science Foundation under CCF-1703489 and by the MGA-2019-42224 project of the Research Fund of Istanbul Technical University, Turkey.



1 Introduction

The suffix tree is arguably the central data structure in Stringology. Briefly speaking, the suffix tree (ST) of a text $T[1,n]$ over an alphabet $\sum = [\sigma] \cup \{\$\}$ is a compact trie over all suffixes, where \$ is the unique terminal symbol. Its linear time construction [10, 25, 32, 33] and efficient tree-navigational features make it a versatile tool in the design of various string matching algorithms. As a practical alternative, suffix arrays were introduced later. Probably the greatest beneficiary of these data structures is bioinformatics; in fact, it is safe to say that the field would not have been the same without them [1, 31]. We refer to Gusfield's book [18] for an exhaustive list of algorithms aided by suffix trees and suffix arrays.

In the era of data deluge, a negative aspect of suffix trees and suffix arrays is their memory footprint of $\Theta(n)$ words or $\Theta(n \log n)$ bits. In comparison, the text can be encoded in just $n \log \sigma$ bits, or even lower space using compression techniques. To put this into perspective, the suffix tree takes around 15 bytes per character and the suffix array takes around 4 bytes per character for human genome, where σ is 4. Bridging the complexity gap between data-space and index-space has been a challenging task. The advent of succinct data structures [19] and compressed text indexing, where the goal is to have a data structure in space close to the information theoretical minimum, presented us with new indexes like the FM–Index by Ferragina and Manzini [12] and the Compressed Suffix Array (CSA) by Grossi and Vitter [17]; these indexes encapsulate the functionalities of suffix array in near-optimal number of bits (w.r.t. statistical entropy). While the CSA achieved this goal via the structural properties of suffix trees/arrays, FM-Index relied on the Burrows-Wheeler Transformation (BWT) of the text [7]. Moreover, the FM-index is a self-index, i.e., any portion of the original text can be extracted from the index. These remarkable breakthroughs saved orders of magnitude of space in practice and eventually became the foundations of more advanced indexes [6, 11, 26, 27, 29, 30]. They are the backbone of many widely used bioinformatics tools like the BWA [22], SOAP2 [24], Bowtie [21], etc.

Motivated by the fact that two human genomes differ in hardly 0.1% of their positions, Belazzougui et al. [5] introduced the concept of Relative Compressed Indexes or Reusable-Indexes, where the objective is to leverage the fact that a full text index (say an FM-index) of a string T is already available, while indexing a "closely similar" string T'. They showed that the FM-index of T' can be encoded in $O(\delta)$ extra space (in words), assuming that the FM-index of T is accessible. Here, δ denotes the edit distance between the 's of T and T'. We study a special, but useful instance of this problem, in which T' is the reverse of T.

1.1 Relative Compression of the Reverse Suffix Array

Let $T[1,n] = t_1 t_2 \dots t_{n-1}\$$ be a string over the alphabet $\Sigma = [\sigma] \cup \{\$\}$, where the character \$ appears exactly once. The reverse of T is the string $\overleftarrow{\mathsf{T}} = t_{n-1} t_{n-2} \dots t_1\$$. We use the following lexicographic order: $\$ < 1 < 2 < \dots < \sigma$. We use $T[i,j]$ (resp., $\overleftarrow{\mathsf{T}}[i,j]$) to denote the substring of T (resp., $\overleftarrow{\mathsf{T}}$) from position i to j.

The suffix array $\mathsf{SA}[1,n]$ stores the starting positions of the lexicographically arranged suffixes, i.e., $\mathsf{SA}[i] = j$ if the i^{th} lexicographically smallest suffix is $T[j,n]$. The inverse suffix array $\mathsf{ISA}[1,n]$ is defined as: $\mathsf{ISA}[j] = i$ if and only if $\mathsf{SA}[i] = j$. Thus, the suffix array and its inverse can be stored in $\Theta(n)$ words, i.e., $\Theta(n \log n)$ bits. The BWT of T is an array $\mathsf{BWT}[1,n]$ such that $\mathsf{BWT}[i] = T[\mathsf{SA}[i]-1]$, where $T[0] = T[n]$. An FM–Index is essentially a combination of the BWT (with rank-select functionality support via a wavelet tree [16]) and a sampled (inverse) suffix array. Likewise, we can define the suffix array and the inverse suffix array for the reverse text $\overleftarrow{\mathsf{T}}$, denoted by $\overleftarrow{\mathsf{SA}}[1,n]$ and $\overleftarrow{\mathsf{ISA}}[1,n]$, respectively.

▶ **Problem 1.** *Can we decode* $\overleftarrow{\mathsf{SA}}[\cdot]$ *and* $\overleftarrow{\mathsf{ISA}}[\cdot]$ *values efficiently using the FM-index of* T*?*

1.2 Motivation and Related Work

We observe that when the application mandates performing search both in forward and reverse directions and we already have an index on the forward text, it is possible to calculate the SA (or ISA) values of the reversed text on the fly efficiently by using the forward index, which eliminates the overhead of reverse suffix array. Some text processing applications, particularly in computational biology, are a good example of this case. For instance, the *read mapping* problem [23] in bioinformatics aims to match a given read onto a reference genome. Due to the DNA sequencing technology used, a read may originate from a forward strand as well as the reverse strand of the DNA helix, and the direction is unknown at the time of mapping. Thus, while the read can be aligned to the reference in its original form, its reverse complement should also be considered as it could be sampled from the reverse strand. One way to cope with this problem is to create two indexes [22], one for the forward and the other for the reverse strand mapping, which obviously doubles the space. However, if the forward index can be used to search in the reverse text, the space can be reduced significantly.

The practical applicability of our study addresses this case by showing that we can compute the $\overleftarrow{\mathsf{SA}}[i]$ and $\overleftarrow{\mathsf{ISA}}[i]$ elements of the reverse text for any possible i, by solely using the FM-Index constructed over the forward text. In a wider sense, any bioinformatics application that makes use of a FM-Index while performing a pattern search on a target sequence, can benefit from our solution to search on the reverse strand of the target without any need of extra space. For example, Lam et al. [20] use both the forward and backward BWT to find matches with k-mismatches allowed; our results eliminate the requirement of the latter, thereby roughly halving the space. It is noteworthy that other relevant elements of the reverse text, such as computing the longest common prefix of two suffixes and $\overleftarrow{\mathsf{BWT}}$-entry can be generated from the $\overleftarrow{\mathsf{SA}}[i]$ and $\overleftarrow{\mathsf{ISA}}[i]$, becomes *efficiently* computable on the fly.

From a theoretical perspective, one can argue that pattern matching on the reverse text is equivalent to matching the reverse of the pattern in the forward text. However, there are applications, where one needs to find the range of suffixes in the suffix tree/array of the reverse text that are prefixed by a pattern. A typical example is the classic solution for approximate pattern matching with one error, which uses the suffix tree/array of the text as well as that of the reverse of the text, along with an orthogonal range searching data structure [2]. A similar approach is followed in most of the compressed indexes based on LZ-compression, although the forward/reverse suffixes arrays/trees are sparse [3]. Another use is in the (relative) compressed indexing of a collection of sequences that are highly similar. Here, two full text indexes corresponding to the reference sequence and its reverse are maintained. Other sequences are indexed in relative LZ compressed space w.r.t. the reference sequence [9]. On a related note, Ohlebusch et al. [28] provided a procedure to compute the BWT of the reverse text considering the strong correlation between T and $\overleftarrow{\mathsf{T}}$. They compute the reverse BWT from the forward BWT, but in their process to compute the k^{th} entry of the BWT, one has to decode all entries from 1 to $k-1$. Their technique can also partially fill the reverse suffix array during this computation, where additional effort is required to calculate the missing elements of $\overleftarrow{\mathsf{SA}}$. Our approach on the other hand can directly compute any BWT entry for the reverse text. In another work, Belazzougui et al. [4] showed how to represent the bi-directional BWT (i.e., forward and reverse BWT) so that one can perform efficient navigation of the suffix tree in the forward and backward direction; however, to search in the forward direction, their representation again needs space roughly twice that of the FM–Index.

1.3 Our Results

The following is our main contribution in this paper.

▶ **Theorem 2.** *Assuming the availability of the FM–Index of $T[1,n]$ (where* BWT *is stored in the form of a wavelet tree), we can compute the suffix array value $r = \overleftarrow{\mathsf{SA}}[i]$ for any given i (resp. inverse suffix array $i = \overleftarrow{\mathsf{ISA}}[r]$ for any given r) of the reversed text $\overleftarrow{T}$ in time $O(h \cdot \mathtt{t}_{\mathtt{WT}} + \mathtt{t}_{\mathtt{SA}})$. Here,*

- *h is the length of the shortest unique substring that starts at position r in $\overleftarrow{\mathsf{T}}$,*
- *$\mathtt{t}_{\mathtt{WT}}$ is the time to support standard wavelet tree operations on the* BWT*, and*
- *$\mathtt{t}_{\mathtt{SA}}$ is the time to decode a suffix array (or inverse suffix array) value using FM–Index.*

In the most common implementations of the FM–Index, $\mathtt{t}_{\mathtt{WT}} = O(\log \sigma)$ and $\mathtt{t}_{\mathtt{SA}} = O(\log^{1+\epsilon} n)$, where $\epsilon > 0$ is arbitrarily small. On average h can be expected to be $O(\log_\sigma n)$ when the text is assumed to be independently and identically distributed over the alphabet Σ [8]. Thus, we get the following corollary.

▶ **Corollary 3.** *Given the FM–Index of T (where* BWT *is stored in the form of a wavelet tree), we can decode a suffix array value or inverse suffix array value of the reversed text in $O(\log^{1+\epsilon} n)$ expected time, where $\epsilon > 0$ is arbitrarily small.*

We complement the above results with experiments. Since Corollary 3 may not hold on sequences with skewed symbol distributions such as natural language texts, we also include such cases in the experiments to analyze the performance. The experiments show that our results are competitive when compared to the standard approach of maintaining the FM–Index for both the forward text and the reverse text.

2 Burrows-Wheeler Transform and FM–Index

Given an array $A[1,m]$ over an alphabet Σ of size σ, by using the wavelet tree data structure of size $m \log \sigma + o(m)$ bits, the following queries can be answered in $O(\log \sigma)$ time [12, 14, 16]:

- $A[i]$,
- $\mathsf{rank}_A(i,j,x)$ = the number of occurrences of x in $A[i,j]$,
- $\mathsf{select}_A(i,j,k,x)$ = the k^{th} occurrence of x in $A[i,j]$,
- $\mathsf{quantile}_A(i,j,k)$ = the k^{th} smallest character in $A[i,j]$,
- $\mathsf{rangeCount}_A(i,j,x,y)$ = the number of positions $k \in [i,j]$ satisfying $x \leq A[k] \leq y$

Burrows and Wheeler [7] introduced a reversible transformation of the text, known as the Burrows-Wheeler Transform (BWT). Let T_x be the circular suffix starting at position x, i.e., $T_1 = T$ and $T_x = T[x,n] \circ T[1,x-1]$, where $x > 1$ and $\circ$ denotes concatenation. Then, the BWT of T is obtained as follows: first create a conceptual matrix M, such that each row of M corresponds to a unique circular suffix, and then lexicographically sort all rows. Thus the ith row in M is given by $T_{\mathsf{SA}[i]}$. The BWT is the last column L of M, i.e., $\mathsf{BWT}[i] = T_{\mathsf{SA}[i]}[n] = T[\mathsf{SA}[i]-1]$, where $T[0] = T[n] = \$$. The main component of the FM–Index is the last-to-first column mapping (in short, LF mapping). For any $i \in [1,n]$, $\mathsf{LF}(i)$ is the row j in the matrix M where $\mathsf{BWT}[i]$ appears as the first character in $T_{\mathsf{SA}[j]}$. Specifically, $\mathsf{LF}(i) = \mathsf{ISA}[\mathsf{SA}[i]-1]$, where $\mathsf{SA}[0] = \mathsf{SA}[n]$.

To compute $\mathsf{LF}(i)$, we store a wavelet tree over $\mathsf{BWT}[1,n]$ in $n \log \sigma + o(n \log \sigma)$ bits. Let the number of occurrences of symbol $i \in \Sigma \setminus \{\$\}$ in T be f_i. We store another array $\mathsf{C}[1,\sigma]$ such that $\mathsf{C}[i]$ is the number of characters in T that are lexicographically smaller than i. Specifically, $\mathsf{C}[1] = 1$, and $\mathsf{C}[i] = 1 + \sum_{j<i} f_j$ when $i > 1$. As a convention, we denote

$\mathsf{C}[\$] = 0$ and $\mathsf{C}[\sigma+1] = n$. Using these, we can compute $\mathsf{LF}(i)$ mapping in $O(\log \sigma)$ time as $\mathsf{LF}(i) = \mathsf{C}[\mathsf{BWT}[i]] + \mathsf{rank}(1, i, \mathsf{BWT}[i])$. We can decode $\mathsf{SA}[i]$ in $O(\log^{1+\epsilon} n)$ time by using LF mapping and by maintaining a sampled-suffix array, which occupies $o(n \log \sigma)$ bits in total. The idea is to explicitly store $(i, \mathsf{SA}[i])$ pairs for all $\mathsf{SA}[i] \in \{1, 1+\Delta, 1+2\Delta, \dots\}$, where $\Delta = \lceil \log_\sigma n \log^\epsilon n \rceil$. The space needed is $O(\frac{n}{\Delta} \log n) = o(n \log \sigma)$ bits. Then, $\mathsf{SA}[i]$ can be obtained directly if the value has been explicitly stored; otherwise, it can be computed via at most Δ number of LF mapping operations in time $O(\Delta \cdot \log \sigma) = O(\log^{1+\epsilon} n)$. We can also decode $\mathsf{ISA}[\cdot]$ using the sampled array in $O(\log^{1+\epsilon} n)$ time.

3 The Method

A substring $T[a, b]$ of T is unique if a is the only occurrence of $T[a, b]$ in T. Note that the unique substring starting at a position a is always defined (since T ends in \$). Moreover the reverse of $T[a, b]$ is also unique in $\overleftarrow{\mathsf{T}}$, and it ends at the position $(n - a + 1)$ in $\overleftarrow{\mathsf{T}}$.

3.1 Decoding $\overleftarrow{\mathsf{SA}}[i]$ for a given i

Our algorithm hinges on the following main lemma.

▶ **Lemma 4.** *Given the FM–Index of T (where the BWT is equipped with a wavelet tree), we can compute the shortest unique substring $\overleftarrow{\mathsf{SUS}}$ in $\overleftarrow{\mathsf{T}}$ starting at $\overleftarrow{\mathsf{SA}}[i]$ in $O(h \cdot \mathtt{t_{WT}})$ time, where $h = |\overleftarrow{\mathsf{SUS}}|$. We can then compute $r = \overleftarrow{\mathsf{SA}}[i]$ in $O(\mathtt{t_{SA}})$ time.*

Proof of Lemma 4. Our task is to compute $r = \overleftarrow{\mathsf{SA}}[i]$ for some i, where h is the length of the shortest unique substring $\overleftarrow{\mathsf{SUS}} = \overleftarrow{\mathsf{T}}[r] \circ \overleftarrow{\mathsf{T}}[r+1] \circ \cdots \circ \overleftarrow{\mathsf{T}}[r+h-1]$ of $\overleftarrow{\mathsf{T}}$ starting at position r. Let the range $[\alpha_k, \beta_k]$ be such that for any $j \in [\alpha_k, \beta_k]$ the suffix $T[\mathsf{SA}[j], n]$ starts with the string $\overleftarrow{\mathsf{T}}[r+k-1] \circ \overleftarrow{\mathsf{T}}[r+k-2] \circ \cdots \circ \overleftarrow{\mathsf{T}}[r]$. Moreover, let q_k be such that $\overleftarrow{\mathsf{T}}[r+k]$ is the q_kth smallest character in $\mathsf{BWT}[\alpha_k, \beta_k]$.

Our idea is to successively compute the ranges $[\alpha_1, \beta_1], [\alpha_2, \beta_2], \dots$ and $q_1, q_2, \dots$ until we get a range $[\alpha_h, \beta_h]$ that contains exactly one suffix, i.e., $\alpha_h = \beta_h$. At each step, we are going to decode the characters $\overleftarrow{\mathsf{T}}[r], \overleftarrow{\mathsf{T}}[r+1], \dots, \overleftarrow{\mathsf{T}}[r+h-1]$. Clearly, $\overleftarrow{\mathsf{SUS}} = \overleftarrow{\mathsf{T}}[r] \circ \overleftarrow{\mathsf{T}}[r+1] \circ \cdots \circ \overleftarrow{\mathsf{T}}[r+h-1]$, and the starting position of SUS in T is $\mathsf{SA}[\alpha_h]$. Therefore,

$$r = n - (\mathsf{SA}[\alpha_h] + h - 1) = n - \mathsf{SA}[\alpha_h] - h + 1$$

We now present the details, starting with the following simple observation. The i^{th} lexicographic suffix of $\overleftarrow{\mathsf{T}}$ starts with the same character as the i^{th} lexicographic suffix of T. Therefore, $\overleftarrow{\mathsf{T}}[r] = T[\mathsf{SA}[i]]$, which is essentially the i^{th} smallest character in $\mathsf{BWT}[1, n]$, and is given by $\mathsf{quantile}(1, n, i)$. Now, we find the range $[\alpha_1, \beta_1]$ in constant time using the array C and $\overleftarrow{\mathsf{T}}[r]$. The next step is to decode the character $\overleftarrow{\mathsf{T}}[r+1]$, and compute the range $[\alpha_2, \beta_2]$. Note that $\overleftarrow{\mathsf{T}}[r, n]$ is the $(i - \alpha_r + 1)^{th}$ lexicographically smallest suffix that starts with $\overleftarrow{\mathsf{T}}[r]$. In other words, $\overleftarrow{\mathsf{T}}[r+1]$ is exactly the $(i - \alpha_r + 1)^{th}$ smallest character in $\mathsf{BWT}[\alpha_1, \beta_1]$. Therefore $q_1 = (i - \alpha_r + 1)$.

The next steps are to decode the character $\overleftarrow{\mathsf{T}}[r+1]$ and compute $[\alpha_2, \beta_2]$, then decode $\overleftarrow{\mathsf{T}}[r+2]$ and compute $[\alpha_3, \beta_3]$, and so on. To do so, we rely on the following recursions. From the definition, $\overleftarrow{\mathsf{T}}[r+k] = \mathsf{quantile}(\alpha_k, \beta_k, q_k)$ for any $k \geq 1$. We now show how to compute $[\alpha_{k+1}, \beta_{k+1}]$. Let a be the smallest index $\geq \alpha_k$ and let b be the largest index $\leq \beta_k$, such that $\mathsf{BWT}[a] = \mathsf{BWT}[b] = \overleftarrow{\mathsf{T}}[r+k]$.

$$\alpha_{k+1} = \mathsf{LF}(a) = \mathsf{C}[\overleftarrow{\mathsf{T}}[r+k]] + \mathsf{rank}(1, \alpha_k - 1, \overleftarrow{\mathsf{T}}[r+k]) + 1$$
$$\beta_{k+1} = \mathsf{LF}(b) = \mathsf{C}[\overleftarrow{\mathsf{T}}[r+k]] + \mathsf{rank}(1, \beta_k, \overleftarrow{\mathsf{T}}[r+k])$$

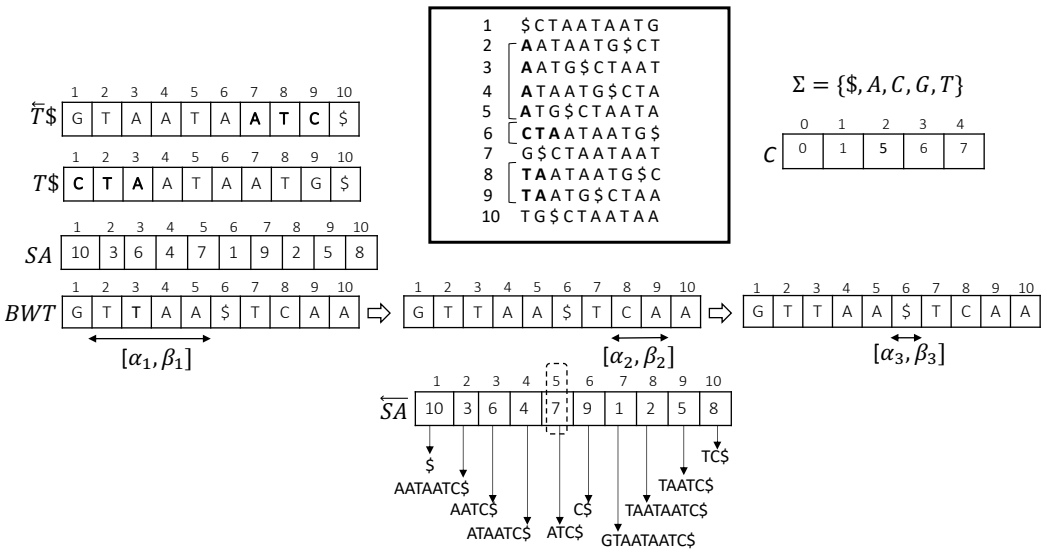


Figure 1 Computing $\overleftarrow{\mathsf{SA}}[5] = 7$ via Lemma 4. Here $\overleftarrow{\mathsf{SUS}} = ATC$, the suffix range $[\alpha_1, \beta_1]$ of A is $[2, 5]$, the suffix range $[\alpha_2, \beta_2]$ of TA is $[8, 9]$, and the suffix range $[\alpha_3, \beta_3]$ of CTA is $[6, 6]$.

Finally, $q_{k+1} = (q_k - d)$, where d is the number of characters in $\mathsf{BWT}[\alpha_k, \beta_k]$ that are lexicographically smaller than $\overleftarrow{\mathsf{T}}[r + k]$, which can be computed via a $\mathsf{rangeCount}$ query.

We repeat this process until we reach $[\alpha_h, \beta_h]$, where $\alpha_h = \beta_h$. This takes $O(h \cdot \mathtt{t}_{\mathtt{WT}})$ time. Then r is decoded in additional $O(\mathtt{t}_{\mathtt{SA}})$ time. This completes the proof of Lemma 4. ◀

Algorithm 1 for computing $\overleftarrow{\mathsf{SA}}[i]$.

```
1:  procedure COMPUTE SA←[i]
2:      if (i = 1) then return n
3:      α ← 1, β ← n, q ← i, h ← 0
4:      while (α < β) do
5:          c ← quantile(α, β, q)
6:          if (q ≠ $) then q ← q − rangeCount(α, β, 1, c − 1)
7:          if (α > 1) then α ← C[c] + rank(α − 1, c) + 1
8:          else α ← C[c] + 1
9:          β ← C[c] + rank(β, c)
10:         h ← h + 1
11:     return n − SA[α] − h + 1
```

3.2 Decoding $\overleftarrow{\mathsf{ISA}}[r]$ for a given r

To compute $\overleftarrow{\mathsf{ISA}}[r]$ for some position r, the main intuition is as follows. Let γ_1 be the number of entries in $\mathsf{BWT}[1, n]$ that are lexicographically smaller than $\overleftarrow{\mathsf{T}}[r]$. Then, $\overleftarrow{\mathsf{ISA}}[r] \geq \gamma_1 = \mathsf{C}[\overleftarrow{\mathsf{T}}[r]]$. Now consider the range $[\alpha_1, \beta_1]$ such that for any $j \in [\alpha_1, \beta_1]$, the suffix $T[\mathsf{SA}[j], n]$ starts with $\overleftarrow{\mathsf{T}}[r]$. Let γ_2 be the number of entries in $\mathsf{BWT}[\alpha_1, \beta_1]$ that are lexicographically smaller than $\overleftarrow{\mathsf{T}}[r + 1]$. Then, $\overleftarrow{\mathsf{ISA}}[r] \geq \gamma_1 + \gamma_2$. Now, consider the range $[\alpha_2, \beta_2]$ such that for

any $j \in [\alpha_2, \beta_2]$, the suffix $T[\mathsf{SA}[j], n]$ starts with $\overleftarrow{\mathsf{T}}[r+1] \circ \overleftarrow{\mathsf{T}}[r]$. Compute γ_3. We repeat the process until we reach the range $[\alpha_h, \beta_h]$ such that $\alpha_h = \beta_h$. Clearly, the unique suffix $T[\mathsf{SA}[\alpha_h], n]$ starts with $\mathsf{SUS} = \overleftarrow{\mathsf{T}}[r+h-1] \circ \overleftarrow{\mathsf{T}}[r+h-2] \cdots \circ \overleftarrow{\mathsf{T}}[r]$. Since SUS is the smallest unique prefix of $\overleftarrow{\mathsf{T}}[r, n]$, $\sum_{s \leq h} \gamma_s$ is the number of suffixes in $\overleftarrow{\mathsf{T}}$ that are lexicographically smaller than $\overleftarrow{\mathsf{T}}[r, n]$. Thus, $\overleftarrow{\mathsf{ISA}}[r] = 1 + \sum_{s \leq h} \gamma_s$. To compute the $\gamma_1, \gamma_2, \ldots, \gamma_h$, we use $\mathsf{rangeCount}$ operation. Computing the range $[\alpha_k, \beta_k]$ from $[\alpha_{k-1}, \beta_{k-1}]$ is achieved using the array C and rank operation, as in proof of Lemma 4.

The algorithm has $h = |\mathsf{SUS}|$ rounds. Each round comprises of a constant number of wavelet tree operations, and accesses to the C array. Additionally, in the k^{th} round, we have to decode the character $\overleftarrow{\mathsf{T}}[r+k-1]$. To do this we use the following technique. If $r = n$, then $\overleftarrow{\mathsf{T}}[r] = \$$; so, assume otherwise. Note that $\overleftarrow{\mathsf{T}}[r] = T[n-r+1]$; thus, $\overleftarrow{\mathsf{T}}[r] = \mathsf{BWT}[\mathsf{ISA}[n-r+2]]$ is found in $O(\mathtt{t_{SA}} + \mathtt{t_{WT}})$ time. Now, $\overleftarrow{\mathsf{T}}[r+1], \overleftarrow{\mathsf{T}}[r+2], \ldots$ are given by $\mathsf{BWT}[\mathsf{LF}(\mathsf{ISA}[n-r+2])], \mathsf{BWT}[\mathsf{LF}(\mathsf{LF}(\mathsf{ISA}[n-r+2]))], \ldots$, in $O(\mathtt{t_{WT}})$ time for each k. Hence, the time taken to compute $\overleftarrow{\mathsf{ISA}}[r]$ is $O(\mathtt{t_{SA}} + h \cdot \mathtt{t_{WT}})$. We have the following lemma.

▶ **Lemma 5.** *Given the FM–Index of T (where the BWT is equipped with a wavelet tree), we can compute $i = \overleftarrow{\mathsf{ISA}}[r]$ in $O(\mathtt{t_{SA}} + h \cdot \mathtt{t_{WT}})$ time, where h is the length of the shortest unique substring of $\overleftarrow{\mathsf{T}}$ that starts at r.*

From Lemma 4 and Lemma 5, Theorem 2 is immediate. We outline Lemmas 4 and 5 formally in Algorithms 1 and 2 respectively. ◀

Algorithm 2 for computing $\overleftarrow{\mathsf{ISA}}[r]$.

```
1:  procedure Compute ISA←[r]
2:      if (r = n) then return 1
3:      i ← ISA[n − r + 2], c ← BWT[i]
4:      α ← 1, β ← n, γ ← C[c]
5:      while (α < β) do
6:          if (α > 1) then α ← C[c] + rank(α − 1, c) + 1
7:          else  α ← C[c] + 1
8:          β ← C[c] + rank(β, c)
9:          i ← LF(i), c ← BWT[i]
10:         if (c ≠ $) then
11:             γ ← γ + rangeCount(α, β, 1, c − 1)
12:     return (1 + γ)
```

4 Experimental Results

The proposed algorithms eliminate the necessity to separately maintain the SA and ISA of the reverse text $\overleftarrow{\mathsf{T}}$ by computing $\overleftarrow{\mathsf{SA}}[i]$ and $\overleftarrow{\mathsf{ISA}}[i]$ directly from the FM-index of T. The natural question is how the performance of the introduced method is compared with the regular access via the FM-index that could be built on $\overleftarrow{\mathsf{T}}$. We have implemented the proposed

algorithms[1] by using the `sdsl-lite` framework[2] [15] and performed some tests on 50MB `dna`, `english`, `proteins`, `sources`, and `dblp.xml` files from `Pizza&Chili`[3] corpus to analyze the practical performance of the introduced methods.

For each file, we have created the FM-index and measured the elapsed time of our algorithm to retrieve $\overleftarrow{\mathsf{SA}}[i]$ / $\overleftarrow{\mathsf{ISA}}[i]$ for 100K randomly selected distinct i positions. We benchmark that elapsed time against a regular SA / $\overleftarrow{\mathsf{ISA}}$ access on the FM-index created over $\overleftarrow{\mathsf{T}}$, assuming that both forward and reverse FM-indices apply the same sampling strategy with the same sampling frequency,

All operations in Algorithms 1 and 2, namely the $\mathsf{quantile}$, $\mathsf{rangeCount}$, rank queries, `backwards_search`, and $\mathsf{LF-mappings}$, are achieved in logarithmic time. The execution times of the introduced algorithms are directly proportional to the number of times they are repeated, which is determined by the length of the matching SUS. Hence, on positions where the SUS is extremely long, the execution time will increase. It makes sense to define a threshold such that the proposed methods stay compatible in practice. Therefore, for those positions that have a SUS longer than this threshold, it may be preferred to pre–compute and maintain their $\overleftarrow{\mathsf{SA}}$/$\overleftarrow{\mathsf{ISA}}$ values offline. We suggest to set this threshold to the SA/ISA sampling frequency used in the FM-index construction. While selecting the random positions in the experiments, those with SUS lengths longer than the threshold are excluded. Table 1 lists the average SUS length of the 100K randomly selected positions with this restriction on each file for each SA sampling frequency. The percentage of all positions that has a shorter SUS than the corresponding sampling frequencies, are also presented.

Table 1 The average SUS lengths of the selected positions on each file per each sampling frequency, and the percentage of all positions in that file, which has SUS length less than or equal to that sampling frequency.

	Average SUS Length			Positions with SUS length $\leq$ Sampling Frequency		
Sampling Frequency:	32	64	128	32	64	128
dblp.xml	19.04	29.34	41.37	58.57%	81.18%	96.81%
dna	15.22	16.46	17.11	96.91%	99.45%	99.89%
english	12.89	14.17	17.48	97.81%	99.05%	99.64%
protein	8.48	11.04	17.23	84.58%	87.80%	91.11%
sources	16.13	21.63	28.10	86.11%	93.36%	96.37%

The experiments were run on an iMac using MacOS 10.13.16 and equipped with 16GB memory and 3.23 GHz Intel Core i5 processor. The software was compiled with the clang LLVM compiler with full optimization (`-O3`). During the experiments, we considered the sampling factors of 32, 64 and 128, along with both `text-ordered` and `suffix-ordered` sampling strategies [13].

The shape of the wavelet tree (WT) representing the BWT may also be an important factor in practical performance to achieve the queries we use in our algorithms. The `lex_ordered(i,j,c)` function of the sdsl-lite platform, which returns the number of symbols lexicographically smaller/greater than `c` in the `(i,j)` interval of a wavelet tree `WT`, is used in

[1] The implementation is available online at `https://github.com/oguzhankulekci/reverseSA`.
[2] The sdsl-lite framework is available online at `https://github.com/simongog/sdsl-lite`.
[3] `http://pizzachili.dcc.uchile.cl/index.html`.

the implementation. This function requires the `WT` to support lexicographical ordering[4], where Hu-Tucker and balanced `WT`s are the only options, and thus, included in the experiments. The Huffman–shaped WT is not used as it does not support the lexicographical ordering.

Figure 2 represents the comparison of the average elapsed time to retrieve the $\overleftarrow{\mathsf{SA}}[i]$ with Algorithm 1 versus the regular access via FM-index of $\overleftarrow{\mathsf{T}}$ on `english`, `dna`, and `protein` files[5], whose alphabet sizes are respectively 239, 16, and 27. Total time to retrieve the $\overleftarrow{\mathsf{SA}}[i]$ via the Algorithm 1 is equal to the sum of the SUS extraction and SA access on forward FM-index. It is observed that `Hu-Tucker` shaped WT provides better running time than the `balanced` shaped and, `text-order` based sampling is superior to the `suffix-order` based. The average SA access time on both forward and reverse directions are approximately equal. Therefore, the expected latency in the proposed technique depends on the SUS-detection phase. As shown in Table 1, due to the limitation applied in the selection process, the average SUS length is increasing as the sampling frequency gets larger. This reflects on the SUS extraction cost in Figure 2, where the SUS extraction time expands proportionally by the increment of the average SUS length in each data type.

Table 2 The ratios representing the overall execution time of the Algorithms 1 and 2 divided by the regular SA and ISA access on different sampling ratios and strategies.

	$\overleftarrow{\mathsf{SA}}$ Benchmark (Algorithm 1)						$\overleftarrow{\mathsf{ISA}}$ Benchmark (Algorithm 2)					
Sampling Strategy:	Text-ordered			Suffix-ordered			Text-ordered			Suffix-ordered		
Sampling Frequency:	32	64	128	32	64	128	32	64	128	32	64	128
dblp.xml	7.0	5.2	3.5	5.0	3.4	2.5	9.5	6.2	3.9	9.4	6.0	3.8
dna	4.2	2.7	1.9	2.7	2.0	1.5	5.1	3.5	2.1	5.5	3.2	2.2
english	4.3	3.0	2.0	2.7	2.0	1.6	5.3	3.3	2.4	5.3	3.2	2.4
protein	2.7	2.1	1.7	1.9	1.6	1.4	3.4	2.5	2.0	3.4	2.5	2.1
sources	4.8	3.4	2.5	3.2	2.4	1.8	6.0	4.1	2.9	6.0	4.1	2.9

With the aim of having a better understanding about the running time of Algorithms 1 and 2, the elapsed time to access a random $\overleftarrow{\mathsf{SA}}[i]$ (and $\overleftarrow{\mathsf{ISA}}[i]$), is divided by the time required to achieve these queries with a regular FM-index constructed over $\overleftarrow{\mathsf{T}}$. Table 2 lists these ratios. Since the proposed methods can retrieve the $\overleftarrow{\mathsf{SA}}[i]$ and $\overleftarrow{\mathsf{ISA}}[i]$ values without the FM-index on $\overleftarrow{\mathsf{T}}$, a slow-down is actually expected in the general paradigm of time-memory trade-off. On `dna` sequences, Algorithm 1 is only 2.7, 2.0, and 1.5 times slower for corresponding sampling frequencies, while using Suffix-ordered method. On `protein` sequences, the ratios are even better to be 1.9, 1.6, and 1.4 respectively. We observed the worst results on `dblp.xml` file, which is highly repetitive, and thus, the SUS extraction times have been observed to be significantly longer. It's reasonable to particularly underline the performance of our proposed algorithm on biological sequences. Since, text operations on reverse direction are expected to be a more common demand in terms of computational biology applications. The proposed solution, especially the $\overleftarrow{\mathsf{SA}}$ calculation, competes better in suffix-based SA sampling strategy. This favours the practical applicability of our theoretical results since suffix-based approach is the default choice in practice due to its space conservation. The $\overleftarrow{\mathsf{ISA}}$ computations with Algorithm 2 are generally observed to be ≈ 1.5 times worse than the $\overleftarrow{\mathsf{SA}}$ computations, which is due to the fact that retrieving ISA is nearly two times faster than accessing SA on an FM-index with equal SA and ISA sampling ratios[6].

[4] Indicated as `lex_ordered` in `http://simongog.github.io/assets/data/sdsl-cheatsheet.pdf`
[5] The `sources` and `dblp` results are not shown in Figure 2 to save space.
[6] As also considered in `sdsl-lite` framework by setting the default ISA sampling frequency to two times of the SA sampling frequency.

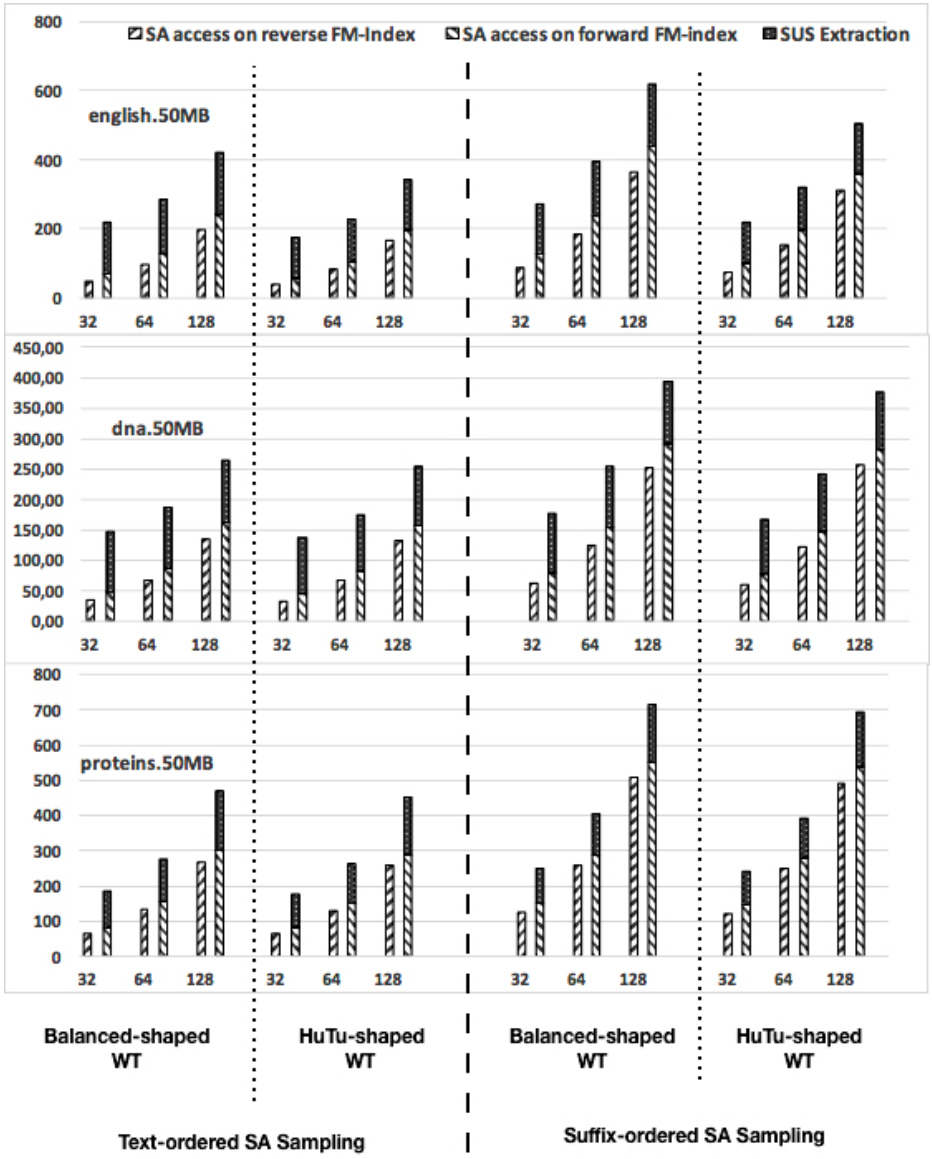


Figure 2 Experimental analysis to compare the speed of the proposed method and the regular SA access on the FM-index constructed over the reversed text. Y-axis represents the elapsed time in microseconds and X-axis indicate the sampling frequencies. For the representation of the BWT, both "`balanced`" and "`Hu-Tucker`" shaped wavelet trees that supports lexicographical ordering (which is required by the methods we use in the algorithms), are considered.

The practical performance of the Algorithms 1 and 2 depends heavily on the length of the corresponding SUS. Short SUSs are expected to be more common, where the method will execute fast. On the other hand, even much less frequently observed longer SUS cases degrade the overall average timings. Thus, for a deeper investigation, the diffraction in Table 3 lists the percentages of the positions on which the elapsed time with the proposed algorithm is "X" times of the regular access on the FM–index constructed over $\overleftarrow{\mathsf{T}}$.

Table 3 Percentage of the positions on which the elapsed time with the proposed algorithms are "X" times of the regular suffix array access on the FM–index constructed over $\overleftarrow{\mathsf{T}}$. For instance, on `protein` sequence, Algorithm 1 answered faster than the regular FM-index over $\overleftarrow{\mathsf{T}}$ on 21.88% of the queries when the sampling factor is 32 with a suffix-ordered strategy. On 40.76% of the cases, the speed is slower than the regular access, but not more than two times. Similarly, the speed is between two and three times of the SA access with FM-index constructed over $\overleftarrow{\mathsf{T}}$ on 21.17% of cases.

	dna			english			protein		
	32	64	128	32	64	128	32	64	128
<1x	0.00	0.03	13.93	0.00	0.30	13.75	0.06	12.92	23.89
1x-2x	0.02	24.69	41.86	0.50	19.42	40.97	25.23	36.87	40.36
2x-3x	12.30	38.71	36.84	10.52	34.57	34.24	35.94	38.83	27.69
3x-4x	31.36	30.13	5.59	27.27	32.14	6.66	30.81	7.04	4.50
>4x	56.32	6.43	1.77	61.70	13.56	4.38	7.96	4.33	3.55
$\overleftarrow{\mathsf{SA}}$ **results with text-ordered sampling strategy**									
<1x	0.03	20.03	41.50	1.37	19.28	38.31	21.88	37.95	45.77
1x-2x	36.46	42.96	34.16	35.97	41.83	33.26	40.76	34.19	31.24
2x-3x	33.49	20.70	14.27	31.76	21.04	15.83	21.17	16.29	14.00
3x-4x	16.22	9.13	5.84	16.10	9.87	7.22	9.28	6.90	5.46
>4x	13.81	7.18	4.24	14.80	7.98	5.38	6.91	4.67	3.53
$\overleftarrow{\mathsf{SA}}$ **results with suffix-ordered sampling strategy**									
<1x	0.00	0.00	5.95	0.00	0.03	5.59	0.00	0.96	10.93
1x-2x	0.00	3.03	42.67	0.02	10.90	36.45	2.14	20.43	36.48
2x-3x	0.11	31.02	41.83	1.57	33.41	38.12	17.93	34.89	36.45
3x-4x	12.10	36.81	6.80	12.63	33.14	12.71	28.05	27.85	9.40
>4x	87.78	29.14	2.75	85.79	22.53	7.14	51.88	15.86	6.75
$\overleftarrow{\mathsf{ISA}}$ **results with text-ordered sampling strategy**									
<1x	0.00	0.00	3.23	0.00	0.03	5.39	0.00	2.26	17.35
1x-2x	0.00	6.99	38.88	0.03	9.46	35.95	4.42	33.26	37.49
2x-3x	0.10	36.14	42.38	2.28	32.59	38.52	32.67	37.99	33.09
3x-4x	10.87	38.00	12.69	14.63	35.07	12.46	36.96	18.20	5.12
>4x	89.03	18.88	2.81	83.06	22.86	7.67	25.95	8.30	6.96
$\overleftarrow{\mathsf{ISA}}$ **results with suffix-ordered sampling strategy**									

Actually, Algorithm 1 already includes a regular SA access in itself (line 11). So, it's quite surprising to observe that there are cases where Algorithm 1 executes faster than the regular access. Such cases appear since the access time to suffix arrays on FM-indices differs in forward and reverse directions. Figure 3, depicts this by sketching the number of accessed symbols with Algorithm 1 and with a regular SA access on reverse FM-index. Algorithm 1 starts with extracting the SUS which ends at k in forward direction (or starts at k in reverse direction). Once X symbols long SUS is extracted, the algorithm calls the regular SA access on the FM-index of T, which tells the location of this SUS on T. This access requires backwards traversal of Y symbols on T via the FM-index, where Y is the distance to the closest sampled point on the left of the SUS. The result of this access is then used to compute the exact value of k on $\overleftarrow{\mathsf{T}}$. On the other hand, when an FM-index of $\overleftarrow{\mathsf{T}}$

(reverse FM-index) is available, Z symbols are subject to backwards traversal (as backwards on $\overleftarrow{\mathsf{T}}$ means left-to-right movement on the Figure 3). When the summation of "SA call on forward FM-index" and the "SUS extraction cost" is smaller than the SA access on the reverse FM-index ($cost(X) + cost(Y) < cost(Z)$), such interesting cases may occur.

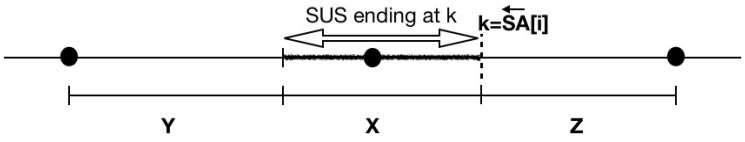


Figure 3 Sketching the number of backwards traversal steps with Algorithm 1 ($X + Y$) versus FM-index of $\overleftarrow{\mathsf{T}}$ (Z). Dark circles represent the sampled positions in both directions.

5 Conclusion

We have presented two algorithms to compute the $\overleftarrow{\mathsf{SA}}[i]$ and $\overleftarrow{\mathsf{ISA}}[i]$ values by using the FM-index of the forward text T. Experiencing slowdown in such space preserving approaches is expected, and hence, we conducted experiments to observe this effect in practice. The benchmark results stated in Table 2 reveals that the $\overleftarrow{\mathsf{SA}}$ and $\overleftarrow{\mathsf{ISA}}$ calculations are respectively 2-3 times and 3-4 times slower on the average when compared to a regular FM-index constructed over $\overleftarrow{\mathsf{T}}$ with suffix-ordered sampling strategy. Particularly on biological sequences, such as the `dna` and `protein` files, the ratios even improve better supporting their usage in practice. Although the execution time of the introduced algorithms increase on sections of long repeats of the input data (as the SUS extraction is the key of the proposed methods), the methods respond quite fast in most cases as shown in Table 3 since the majority of the SUS lengths are centric around shorter lengths. Another interesting application of the proposed methods might be in fully-parallel constructing the BWT of the reverse text from the forward BWT, which has been mentioned in the previous study of Ohlebusch et al. [28] with a solution by computing $\overleftarrow{\mathsf{BWT}}[k]$ under assumption that $\overleftarrow{\mathsf{BWT}}[1], \overleftarrow{\mathsf{BWT}}[2], \ldots, \overleftarrow{\mathsf{BWT}}[k-1]$ are already available, and k iterates from 1 to n. They observed that some positions are independent of the previous ones, which provide an opportunity in parallelizing the execution. However, the level of parallelization here is bounded by the number of such independent start points. Contrary to that, our solution in computing $\overleftarrow{\mathsf{SA}}[i]$ does not introduce any prerequisites for any i, and thus, is fully parallelizable that is scalable up to n processors.

References

1 Srinivas Aluru. *Handbook of Computational Molecular Biology.* Chapman & Hall/CRC, 2005.

2 Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text indexing and dictionary matching with one error. *Journal of Algorithms*, 37(2):309–325, 2000. doi:10.1006/jagm.2000.1104.

3 Diego Arroyuelo, Gonzalo Navarro, and Kunihiko Sadakane. Stronger Lempel–Ziv based compressed text indexing. *Algorithmica*, 62(1-2):54–101, 2012. doi:10.1007/s00453-010-9443-8.

4 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In *Algorithms - ESA 2013 - 21st Annual European Symposium*, pages 133–144, 2013. doi:10.1007/978-3-642-40450-4_12.

5 Djamal Belazzougui, Travis Gagie, Simon Gog, Giovanni Manzini, and Jouni Sirén. Relative FM-indexes. In *String Processing and Information Retrieval - 21st International Symposium,*

SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings, pages 52–64, 2014. doi:10.1007/978-3-319-11918-2_6.

6 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 225–235, 2012. doi:10.1007/978-3-642-33122-0_18.

7 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation (now part of Hewlett-Packard, Palo Alto, CA), 1994.

8 Luc Devroye, Wojciech Szpankowski, and Bonita Rais. A note on the height of suffix trees. *SIAM J. Comput.*, 21(1):48–53, 1992. doi:10.1137/0221005.

9 Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative lempel-ziv self-index for similar sequences. *Theor. Comput. Sci.*, 532:14–30, 2014. doi:10.1016/j.tcs.2013.07.024.

10 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143, 1997. doi:10.1109/SFCS.1997.646102.

11 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1), 2009. *An extended abstract appeared in FOCS 2005 under the title "Structuring labeled trees for optimal succinctness, and beyond"*. doi:10.1145/1613676.1613680.

12 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. *An extended abstract appeared in FOCS 2000 under the title "Opportunistic Data Structures with Applications"*. doi:10.1145/1082036.1082039.

13 Paolo Ferragina, Jouni Sirén, and Rossano Venturini. Distribution-aware compressed full-text indexes. *Algorithmica*, 67(4):529–546, 2013.

14 Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, SPIRE '09, pages 1–6, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-03784-9_1.

15 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 2014)*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.

16 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual Symposium on Discrete Algorithms ACM-SIAM, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 841–850, 2003.

17 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. *An extended abstract appeared in STOC 2000.* doi:10.1137/S0097539702402354.

18 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

19 Guy Joseph Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1988. AAI8918056.

20 Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon C. K. Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional BWT. In *2009 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2009, Washington, DC, USA, November 1-4, 2009, Proceedings*, pages 31–36, 2009. doi:10.1109/BIBM.2009.42.

21 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

22 Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

23 Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.

24 Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: An improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

25 Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

26 Martin D. Muggli, Alexander Bowe, Noelle R. Noyes, Paul S. Morley, Keith E. Belk, Robert Raymond, Travis Gagie, Simon J. Puglisi, and Christina Boucher. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017. doi:10.1093/bioinformatics/btx067.

27 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007. doi:10.1145/1216370.1216372.

28 Enno Ohlebusch, Timo Beller, and Mohamed I Abouelhoda. Computing the Burrows–Wheeler transform of a string and its reverse in parallel. *Journal of Discrete Algorithms*, 25:21–33, 2014.

29 Alessio Orlandi and Rossano Venturini. Space-efficient substring occurrence estimation. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 95–106, 2011. doi:10.1145/1989284.1989300.

30 Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Information and Computation*, 213:13–22, 2012. doi:10.1016/j.ic.2011.03.007.

31 Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction.* Chapman & Hall/CRC, 1st edition, 2009.

32 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

33 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.