Accessing the Suffix Array via ϕ^{-1}-Forest

Christina Boucher[1], Dominik Köppl[2], Herman Perera[1], and Massimiliano Rossi[1]

[1] Department of Computer and Information Science and Engineering,
Herbert Wertheim College of Engineering,
University of Florida, Gainesville, FL, USA
{christinaboucher, rossi.m, hperera1}@ufl.edu

[2] Tokyo Medical and Dental University,
M&D Data Science Center, Tokyo, Japan,
koeppl.dsc@tmd.ac.jp

Abstract. Kärkkainen et al. (CPM, 2009) defined the concept of ϕ that later became key to the construction of the r-index. Given a string $S[1..n]$, its suffix array SA and its inverse suffix array ISA, we define ϕ as the permutation of $\{1, \ldots, n\}$ such that $\phi(i) = \mathrm{SA}[\mathrm{ISA}[i] - 1]$ if $\mathrm{ISA}[i] > 1$, and $\phi(i) = \mathrm{SA}[n]$ otherwise. Gagie et al. (JACM, 2020) showed that it is possible to store $\mathcal{O}(r)$ words such that the permutations ϕ and ϕ^{-1} are evaluated in $\mathcal{O}(\log\log_w(n/r))$-time, which was improved to $\mathcal{O}(1)$-time by Nishimoto and Tabei (ICALP, 2021). In this paper, we introduce the concept of ϕ^{-1}-forest, which is a data structure using sampled SA values to speed up random access to SA. We implemented our approach and compared its performance with respect to the r-index.

Keywords: compressed suffix array, r-index, ϕ function, Burrows–Wheeler transform

1 Introduction

Biological public datasets have become increasingly large, extensive and numerous. Currently, for almost every biomedically or agriculturally interesting species, there exists a sequencing consortium aimed at sequencing a large number of individuals or cultivars of that species. For example, with relative ease you can download over 5,000 human genomes, over 3 million SARS-COVID sequence datasets, and over 5 petabytes sequence data from The Cancer Genome Atlas (TCGA). Fortunately or unfortunately the majority of methods that aim to analyze these and other large datasets rely on the use of succinct data structures that are capable of being constructed and stored in relatively small amount of space and time, while performing efficient queries. In this paper, we focus on developing a data structure that provides efficient access to the *suffix array*, which is defined as follows: $\mathrm{SA}[1..n]$ is an array that is a permutation of $1, \ldots, n$ such that the suffixes of S starting at the consecutive positions indicated in SA are in lexicographical order.

The suffix array, however, predates read alignment or even high-throughput sequencing as it was first introduced by Manber and Myers [12]. The lexicographical order implies that the starting positions of the longest match of a pattern P is within a contiguous range in SA. When combined with the Burrows–Wheeler Transform (BWT) of S [3], this range can be found in $2|P|$ rank queries via backward search. This is one of the main features that read alignment methods, such as Bowtie [9] and BWA [10], exploit in order to efficiently align reads to a database of genomes. Hence, given this application and the ever-increasing size of public, biological datasets, there has been significant interest in reducing the construction size and time of the SA while still allowing for efficient queries. There has been a significant amount of work in developing more efficient compressed suffix array representations and implementations. Most recently Puglisi and Zhukova [15] showed that the suffix array can be compressed via relative Lempel–Ziv (RLZ) dictionary compression in a manner that does not greatly affect the time to access the elements of the SA. In a different direction, Gagie et al. [7] showed that a single element of the SA for each run of the BWT is needed, and that all elements of the SA can be recovered via a small auxiliary data structure (e.g., ϕ). The resulting data structures of Gagie

S = GATTACAT$GATACAT$GATTAGATA#

i	SA	BWT	rotations matrix
0	**26**	A	#GATTACAT$GATACAT$GATTAGATA
1	**8**	T	$GATACAT$GATTAGATA#GATTACAT
2	16	T	$GATTAGATA#GATTACAT$GATACAT
3	25	T	A#GATTACAT$GATACAT$GATTAGAT
4	4	T	ACAT$GATACAT$GATTAGATA#GATT
5	12	T	ACAT$GATTAGATA#GATTACAT$GAT
6	**21**	T	AGATA#GATTACAT$GATACAT$GATT
7	**6**	C	AT$GATACAT$GATTAGATA#GATTAC
8	**14**	C	AT$GATTAGATA#GATTACAT$GATAC
9	**23**	G	ATA#GATTACAT$GATACAT$GATTAG
10	10	G	ATACAT$GATTAGATA#GATTACAT$G
11	1	G	ATTACAT$GATACAT$GATTAGATA#G
12	**18**	G	ATTAGATA#GATTACAT$GATACAT$G
13	**5**	A	CAT$GATACAT$GATTAGATA#GATTA
14	13	A	CAT$GATTAGATA#GATTACAT$GATA
15	**22**	A	GATA#GATTACAT$GATACAT$GATTA
16	**9**	$	GATACAT$GATTAGATA#GATTACAT$
17	**0**	#	GATTACAT$GATACAT$GATTAGATA#
18	**17**	$	GATTAGATA#GATTACAT$GATACAT$
19	**7**	A	T$GATACAT$GATTAGATA#GATTACA
20	15	A	T$GATTAGATA#GATTACAT$GATACA
21	**24**	A	TA#GATTACAT$GATACAT$GATTAGA
22	**3**	T	TACAT$GATACAT$GATTAGATA#GAT
23	**11**	A	TACAT$GATTAGATA#GATTACAT$GA
24	**20**	T	TAGATA#GATTACAT$GATACAT$GAT
25	**2**	A	TTACAT$GATACAT$GATTAGATA#GA
26	**19**	A	TTAGATA#GATTACAT$GATACAT$GA

Table 1: SA, BWT, and *rotations matrix* of the string $S =$ `GATTACAT$GATACAT$GATTAGATA#`. The SA samples at the beginning and end of each run of the BWT are highlighted in bold. We have $r = 13$.

et al. is referred to as the r-index, where r stands for the number of single character runs in the BWT. Then Cobas et al. [5] improved upon the space usage of the r-index by a more careful sampling of the elements of the SA. Their resulting method, referred to as the subsampled r-index or the sr-index, was shown to be between 1.5 to 3.0 times smaller than the r-index in practice.

Although the methods of Gagie et al. and Cobas et al. are provably and practically space efficient, the time required to access the elements of the SA (which relies on ϕ or ϕ^{-1}) is slow in practice. In this paper, we revisit the problem of providing efficient access to the suffix array in the sampled suffix array of Gagie et al. via a small auxiliary data structure that exploits the iterative properties of ϕ. We implemented our method and compared it to the r-index on Chromosome 19 sequences from the 1000 Genomes Project [17] and the Pizza&Chili repetitive corpus [1]. We showed that the runtime for random access of our new method was favorable to that of the r-index. We report our new method was consistently faster on the Pizza&Chili, offering between 3x and 6x speed-up on all Pizza&Chili datasets except for one (`cere`) where both methods performed comparably. Our method is publicly available at `https://github.com/koeppl/rasarindex`.

2 Preliminaries

We define a string S as a finite sequence of characters $S = S[1..n] = S[1] \cdots S[n]$ over an alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$. We denote by ε the empty string, and the length of S as $|S|$. We denote by $S[i..j]$ the substring $S[i] \cdots S[j]$ of S starting in position i and ending in position j, with $S[i..j] = \varepsilon$ if $i > j$. For a string S and $1 \leq i \leq n$, $S[1..i]$ is called the i-th prefix of S, and $S[i..n]$ is called the i-th suffix of S.

Given an array $A[1..n]$ of n integers over a universe $\mathcal{U} = \{1, ..., |\mathcal{U}|\}$, for all $i \in \mathcal{U}$ we define $\text{pred}_A(i)$ as the *predecessor* of i in A, i.e., $\text{pred}_A(i) = \max\{A[j] \leq i \mid 1 \leq j \leq n\}$. Analogously, we define $\text{succ}_A(i)$ as the *successor* of i in A, i.e., $\text{succ}_A(i) = \min\{A[j] \geq i \mid 1 \leq j \leq n\}$.

Next, to define the suffix array for a string S of length n, we consider all rotations of S in lexicographical order. The index of the starting positions in S of each rotation defines the SA of S. Related to the suffix array, given a string S and the suffix array of S, we define the *inverse suffix array* (ISA) of S as the indexes in the SA of every suffix of S, i.e., $\text{ISA}[i] = j$ if and only if $\text{SA}[j] = i$.

In what follows, we assume that S ends with a unique character # smaller than all other characters appearing in S. Then the BWT of S is a permutation of S such that $\mathrm{BWT}[i] = S[\mathrm{SA}[i]-1]$ and $\mathrm{BWT}[i] = \#$ if $\mathrm{SA}[i] = 1$. The BWT is often run-length compressed, i.e., representing maximal consecutive appearances of the same character, also called *runs*, by a single occurrence of this character and an exponent reflecting the length of this run. We denote the number of the runs in BWT by r. See Table 1 for an example.

Throughout this paper, we assume working in the RAM model with machine word size w. Let us fix the length n of a given input string S. Kärkkainen et al. [8] defined the concept of the ϕ function that later became key to the construction of the r-index. Given a string $S[1..n]$, its suffix array SA and its inverse suffix array ISA, we define ϕ as the permutation of $\{1, \ldots, n\}$ such that $\phi(i) = \mathrm{SA}[\mathrm{ISA}[i] - 1]$ if $\mathrm{ISA}[i] > 1$, and $\phi(i) = \mathrm{SA}[n]$ otherwise. Later, Gagie et al. [7] showed that it is possible to store $\mathcal{O}(r)$ words such that the permutations ϕ and ϕ^{-1} are evaluated in $\mathcal{O}(\log\log_w(n/r))$-time. We note that this was later improved by Nishimoto and Tabei [13] to $\mathcal{O}(1)$ time. In particular, let $\mathcal{E}$ be the list of the SA samples at the end of each run of the BWT such that $\mathcal{E}[i]$ is the sample corresponding to the i-th run. Analogously, let $\mathcal{S}$ be the list of the SA samples at the beginning of each run such that $\mathcal{S}[i]$ is the sample corresponding to the i-th run. See Table 2 for an example of $\mathcal{E}$ and $\mathcal{S}$.

Given Gagie et al.'s representation of ϕ, we can implement random access queries to the suffix array by iterating the ϕ function as follows. Let i be a position in the suffix array, and let $j \geq i$ be a sampled position of the suffix array, then $\mathrm{SA}[i] = \phi^{j-i}(\mathrm{SA}[j])$. This solution works in $\mathcal{O}(r)$ space, since $\mathcal{E}$ and $\mathcal{S}$ each contain r SA samples. The running time can be bounded by the length of the longest run.

i	1	2	3	4	5	6	7	8	9	10	11	12	13
$\mathcal{E}[i]$	26	21	14	18	22	9	0	17	24	3	11	20	19
$\mathcal{S}[i]$	26	8	6	23	5	9	0	17	7	3	11	20	2
c_i	5	3	1	2	0	0	0	4	0	0	0	0	2
ℓ_i	1	1	3	1	2	2	3	1	2	6	3	2	

Table 2: Lists $\mathcal{S}$ and $\mathcal{E}$ of our running example with costs c_i and limits ℓ_i. $\mathrm{BWT}[\mathcal{S}[x]..\mathcal{E}[x]]$ is a unary string, for $x \in [1..r]$.

3 Access Data Structures to SA

In what follows, we assume that we have the SA samples of $\mathcal{S}$ and $\mathcal{E}$ as defined by Gagie et al. [7], and describe a method for accessing the missing SA samples via the construction of an auxiliary data structure that exploits the iterative computation of ϕ^{-1}. We note that we selected to describe our methods in terms of ϕ^{-1}, and due to the symmetry of ϕ^{-1} and ϕ everything can be described analogously for ϕ.

The key observation of our technique is that the difference between $\mathrm{SA}[i]$ and $\mathrm{SA}[i+1]$ keeps the same when iteratively exchanging i with the index j such that $\mathrm{SA}[j] = \mathrm{SA}[i] - 1$ as long as $\mathrm{BWT}[i] = \mathrm{BWT}[i+1]$. This is also known as a part of the so-called toehold lemma [6, Lemma 3.1]. In Table 1, when starting with $i = 1$, we have $\mathrm{SA}[i] = 8$ and $\mathrm{SA}[i+1] - \mathrm{SA}[i] = 8$. Then we exchange i with 19 such that $\mathrm{SA}[i] = 7$ but $\mathrm{SA}[i+1] - \mathrm{SA}[i] = 8$. The difference is the same up until $\mathrm{SA}[i] = 3$ with $i = 22$ (for $\mathrm{SA}[j] = 2$ with $j = 25$ we have $\mathrm{SA}[j+1] - \mathrm{SA}[j] = 17$). We can also observe that the number of iterations from SA value 8 to SA value 3 is five, which is the longest common suffix of Rows 1 and 2 of the rotations matrix. In particular, among all SA values stored in $\mathcal{E}$, 3 is the preceding text position of 8. In other terms, this iteration ends whenever we visit the predecessor stored in $\mathcal{E}$ since at that time we know we are at a run boundary and therefore the next value in BWT has to differ. We can make use of this phenomenon for computing ϕ^{-1} as follows. First, having $\mathcal{E}$ and $\mathcal{S}$, we can compute ϕ^{-1} for all SA samples stored in $\mathcal{E}$ because $\phi^{-1}(\mathcal{E}[x]) = \mathcal{S}[x+1]$ for all $x = [1..r-1]$, and $\phi^{-1}(\mathcal{E}[r]) = \mathcal{S}[1]$. Now, given a text position $i \in [1..n]$, let $s = \mathrm{pred}_{\mathcal{E}}(i) \in \mathcal{E}$ be an SA sample for which we can evaluate ϕ^{-1}. Then, by the above observation, the difference between

s and s's succeeding value in SA is the same as the difference of i and i's succeeding value in SA. Hence, $\phi^{-1}(i) = \text{SA}[\text{ISA}[i]+1] = \text{SA}[\text{ISA}[s]+1] + (i-s) = \phi^{-1}(s) + (i-s)$.

3.1 Access via ϕ^{-1}-Graph

We first give two definitions before defining our auxiliary data structure.

Definition 1 (Costs and Limits). *For $i \in [1..r-1]$, we define the* cost $c_i = \mathcal{S}[i+1] - \text{pred}_{\mathcal{E}}(i+1)$, *and the* limit *to be* $\ell_i = \text{succ}_{\mathcal{E}}(\mathcal{E}[i]+1) - \mathcal{E}[i]$.

Informally, costs reflect the distances in our above key observation. Starting with an initial cost of zero, by adding up the costs and moving to elements of $\mathcal{E}$, we can simulate Φ^{-1} as long as the costs are bounded by the limits. The limits reflect the fact that our key observation only works up to the point that the predecessor in $\mathcal{E}$ keeps the same, therefore when the distances become too large, we would move into the next run with predecessor $\text{succ}_{\mathcal{E}}(\mathcal{E}[i]+1)$. Using costs and limits, we define the following directed graph with edge labels.

Definition 2 (ϕ^{-1}-Graphs). *Given $\mathcal{E}$ and $\mathcal{S}$, we build a directed graph $G=(V,E)$ such that there exists a node in G for each sample at the end of each* BWT *run, i.e., $V=\mathcal{E}$, and an edge from v_i (corresponding to $\mathcal{E}[i]$) to v_j (corresponding to $\mathcal{E}[j]$) if and only if $\mathcal{E}[j]$ is the predecessor of $\mathcal{S}[i+1]$ in $\mathcal{E}$. Lastly, we label edge (v_i, v_j) with the cost and limit $< c_i, \ell_i >$. We refer to this as the ϕ^{-1}-*graph.

An example of the ϕ^{-1}-graph is depicted in Figure 1. We observe that all nodes of the ϕ^{-1}-graph have at most one outgoing edge due to the uniqueness of the predecessor function, and the only node with no outgoing edges is the node corresponding to $\mathcal{E}[r]$ since $\mathcal{S}[r+1]$ is not defined. In what follows, we refer to the node of the ϕ^{-1}-graph corresponding to $\mathcal{E}[i]$ as v_i, and the edge outgoing from v_i to v_j as $e_i = (v_i, v_j)$. This induces an injective mapping from edges e_i to nodes v_i (we identify an edge by its outgoing node).

Given the ϕ^{-1}-graph G, we can compute recursive applications of ϕ^{-1} as follows. Given a position $1 \le s \le n$ in the suffix array, let $v_i \in V$ be the node corresponding to $\text{pred}_{\mathcal{E}}(\text{SA}[s])$ and let $e_i = (v_i, v_j)$ be the edge outgoing from v_i, labeled with $< c_i, \ell_i >$. First, we consider the simplest case where $\text{SA}[s]$ is the end of the i-th run. In this case, we can compute ϕ^{-1} directly with the equation

$$\phi^{-1}(\text{SA}[s]) = \text{SA}[s+1] = \mathcal{S}[i+1] = \mathcal{E}[j] + \mathcal{S}[i+1] - \mathcal{E}[j] = \mathcal{E}[j] + c_i.$$

In our example, if we start from the node corresponding to the SA sample 26, by following the edge connecting 26 with 3, we have that $\phi^{-1}(26) = 3 + 5 = 8$.

Next, suppose we want to compute $\phi^{-1}(\text{SA}[s+1])$. Let $e_j = (v_j, v_k)$ be the edge outgoing from v_j, labeled with $< c_j, \ell_j >$. Hence,

$$\phi^{-1}(\text{SA}[s+1]) = \phi^{-1}(\phi^{-1}(\text{SA}[s])) = \phi^{-1}(\mathcal{S}[i+1]) = \phi^{-1}(\mathcal{E}[j] + c_i).$$

If $\text{pred}_{\mathcal{E}}(\mathcal{S}[i+1]) = \mathcal{E}[j]$ then $\phi^{-1}(\mathcal{E}[j] + c_i) = \phi^{-1}(\mathcal{E}[j]) + c_i = \mathcal{E}[k] + c_j + c_i$. This shows that as long as the condition $\text{pred}_{\mathcal{E}}(\mathcal{S}[i+1]) = \mathcal{E}[j]$ holds, we can compute iterated applications of ϕ^{-1} by traversing the graph and summing the costs stored at the edge labels.

Back to our example, we continue at the node corresponding to the SA sample 3, having already gathered a cumulative cost of 5. We now follow the edge connecting 3 with 11, and obtain $\phi^{-1}(8) = \phi^{-1}(3) + 5 = 16$. Note that we cannot compute $\phi^{-1}(16)$ by following the edge connecting 11 and 20, since $\text{pred}_{\mathcal{E}}(16) = 14 \neq 11$.

In the case where $\text{SA}[s]$ is not at the end of a run, let $\mathcal{E}[i] = \text{pred}_{\mathcal{E}}(\text{SA}[s])$. We can write $\phi^{-1}(\text{SA}[s]) = \mathcal{E}[j] + c_i + c_0 = \mathcal{S}[i+1] + c_0$, where $c_0 = \text{SA}[s] - \mathcal{E}[i]$ by definition of ϕ^{-1}. We refer to c_0 as the *initial cost*.

Therefore, given $\text{SA}[s]$, where $\text{pred}_{\mathcal{E}}(\text{SA}[s]) = \mathcal{E}[i_1]$, we let $c_0 = \text{SA}[s] - \mathcal{E}[i_1]$ and $v_{i_1}, v_{i_2}, \ldots, v_{i_m}$ be the m-length path outgoing from v_i. If $\text{pred}_{\mathcal{E}}(\mathcal{E}[i_j] + \sum_{k=1}^{j-1} c_{i_k} + c_0) = \mathcal{E}[i_{j+1}]$ for all $j = 2, ..., m-1$, then after m iterations of ϕ^{-1}, we have

$$\phi^{-m}(\text{SA}[s]) = \mathcal{S}[i_m + 1] = \mathcal{E}[i_m] + \sum_{k=1}^{m-1} c_{i_k} + c_0.$$

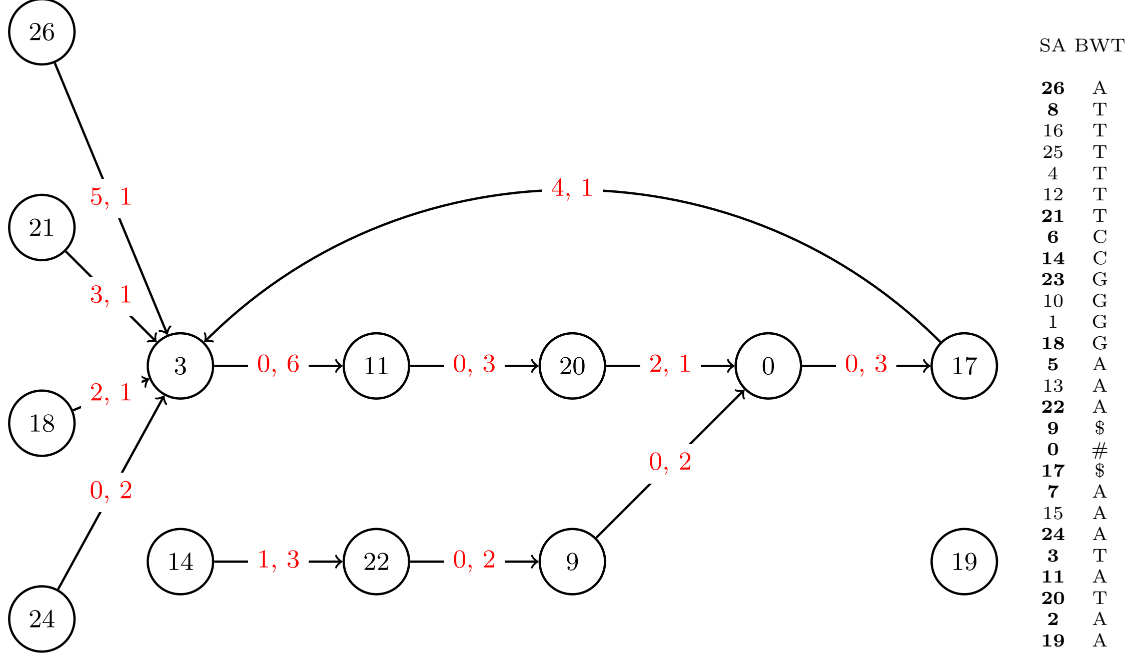

Fig. 1: The ϕ^{-1}-graph corresponding to the string $S = \texttt{GATTACAT\$GATACAT\$GATTAGATA\#}$. Each node is labeled with the SA value at the end of each run. Each edge is labeled with its cost and its limit, which are shown in red. On the right we have the SA and BWT of the string S as derived in Table 1.

We note that checking that $\text{pred}_{\mathcal{E}}(\mathcal{E}[i_j] + \sum_{k=1}^{j-1} c_{i_k} + c_0) = \mathcal{E}[i_{j+1}]$ for all $j = 2, ..., m-1$ is equivalent to checking that

$$\mathcal{E}[i_j] + \sum_{k=1}^{j-1} c_{i_k} + c_0 = \mathcal{S}[i_{j-1} + 1] + c_0 < \text{succ}_{\mathcal{E}}(\mathcal{E}[i_j] + 1) = \mathcal{E}[i_j] + \ell_{i_j},$$

that is equivalent to determining that for all $j = 2, ..., m-1$

$$\sum_{k=1}^{j-1} c_{i_k} + c_0 < \ell_j.$$

Hence, it follows that we can access the SA samples via computing the costs in G with the condition that it is less than the limits at all visited edges.

On our running example, we can follow the edge connecting 3 and 11 since $\text{pred}_{\mathcal{E}}(8) = 3$, which can be obtained checking if the cost 5 is smaller than the limit on the edge that is 6. However, we cannot traverse the edge between 11 and 20 starting with a total cost of 5, since $\text{pred}_{\mathcal{E}}(16) = 14 \neq 11$ which is witnessed by the limit on the edge being 3, which is smaller than the cumulative cost of 5, given by the sum of costs on the edges between 26 and 3, and between 3 and 11.

3.2 Access via ϕ^{-1}-Forest

After we have the ϕ^{-1}-graph G, we can extract (long) paths from G and use them to speed up the computation for random access by allowing us to skip iterations of ϕ^{-1} to find a missing SA sample. We refer to this as *fast-forwarding* the random access. Given a path of length m in G, say $v_{i_1}, v_{i_2}, \ldots, v_{i_m}$, we want to build a data structure that solves the following problem: for a given $1 \leq j \leq m$ and a value c_0, find the largest index k such that $\sum_{x=j}^{h-1} c_{i_x} + c_0 < \ell_h$, for all $h = j, ..., k-1$.

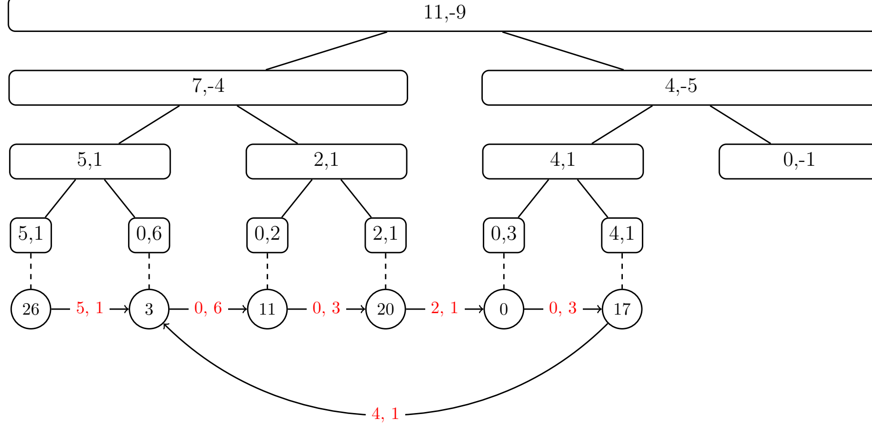

Fig. 2: Example of a balanced binary tree described in Sect. 3.2 on the path $[26, 3, 11, 20, 0, 17, 3]$ of the ϕ^{-1}-graph of Fig. 1. By adding dummy nodes like the rightmost one with label $(0, -1)$, we make the tree full binary. This is not a requirement, but used in our implementation to keep all leaves on the same level such that we can store them consecutively in an array for leveraging data locality. That is because we alternatively can scan this array of leaves instead of walking through the tree if we know that we will only perform a short traversal. The downside is that we need to take care of the dummy nodes. The negative limits $(0, -1)$ of the dummy nodes prevents the algorithm from exploring the non-existent children of this node.

To solve this problem, we build a balanced binary tree, where the leaves represent the edges $e_{i_1}, e_{i_2}, \ldots, e_{i_m-1}$ of the path. Each node p of the tree stores a pair of integers $< c_p, \ell_p >$ such that p is the j-th leaf $< c_p, \ell_p >$ corresponding to the edge label of e_{i_j} if and only if $c_p = c_{i_j}$ and $\ell_p = \ell_{i_j}$. Otherwise, if p is an internal node, we let u and v be the left and right children of p respectively, and $c_p = c_u + c_v$ and $\ell_p = \min(\ell_u, \ell_v - c_u)$. We note that the definition of the costs and limits for the internal nodes guarantee that we can traverse all the edges in the subtree of the node p if we start from the leftmost leaf in the subtree of p with a cost $c < \ell_p$. See Fig. 2 for an example.

Given a suffix array sample $\mathrm{SA}[s]$, we can query the tree as follows. Let i be such that $\mathcal{E}[i] = \mathrm{pred}_{\mathcal{E}}(\mathrm{SA}[s])$, $c_0 = \mathrm{SA}[s] - \mathcal{E}[i]$, and let v be the leaf corresponding to the edge $e_i = e_{i_j}$. In order to find the largest index k such that $\sum_{x=j}^{h-1} c_{i_x} + c_0 < \ell_h$ for all $h = j, ..., k-1$, we divide the query into two phases.

- In the first phase (Line 4 of Algo. 1), we traverse the tree from the leaf v towards the root as follows. We consider a cumulative cost c, which we initialize to c_0. Now let p be the parent of v. If v is a right child of p we move to the parent of p. If v is a left child of p let u be the right child of p. If $c < \ell_u$ then we add c_u to c and we move to the parent of p. Otherwise, if $c \geq \ell_u$ or v is the root, we stop and continue with the second phase.
- In the second phase (Line 13 of Algo. 1), we begin descending the tree starting from the right child u of p. Let q be the left child of u. If $c < \ell_q$ we move to the right child of u and update the value of c to $c + c_q$; otherwise we move to q. We repeat this procedure as long as u is not a leaf. At the end of the procedure we arrive at the leaf corresponding to the edge $e_{i_{k-1}}$. In addition to the node v_{i_k} that corresponds to the sample $\mathcal{E}[i_k]$, we report also the total cost c and the number of traversed edges $d = k - j$ so that we can recover the suffix array sample $\mathrm{SA}[s + d] = \mathcal{E}[i_k] + c$.

The steps of both phases are summarized in Algo. 1 and we refer to them as *fast-forward* query. A trivial case is when all costs are less than the limits; in such a case we would ascend to the root and then descend to the rightmost leaf stored in the tree.

We can easily extend the fast-forward query by including a constraint on the total number of leaves that can be traversed, i.e., solving the following problem: given $1 \leq j \leq m$, a value c_0, and an integer d, find the

Algorithm 1 Computes the largest number of ϕ^{-1} steps in ϕ^{-1}-graph via ϕ^{-1}-forest

```
 1: function FAST-FORWARD(𝒯, v, c_0)
 2:     Let v ∈ 𝒯 be the j-th leaf of 𝒯, i.e., v = v_{i_j}.
 3:     c ← c_0.
 4:     while c < ℓ_v and v is not the root do
 5:         p ← parent(v).
 6:         if v = left-child(p) then
 7:             u ← right-child(p).
 8:             if c < ℓ_u then c ← c + c_u.
 9:             else p ← u.
10:         v ← p.
11:     if v is not leaf then
12:         v ← right-child(v).
13:     while v is not leaf do
14:         q ← left-child(v).
15:         if c < ℓ_q then
16:             c ← c + c_q.
17:             v ← right-child(v).
18:         else v ← q.
19:     Let i_{k-1} be the index of v in 𝒯.
20:     return (i_k, c, k − j)
```

largest index $k \leq d$ such that $\sum_{x=j}^{h-1} c_{i_x} + c_0 < \ell_h$, for all $h = j, ..., k-1$. We refer to this function as *bounded fast-forward*. Thus, applying the above procedures to perform SA access is straightforward when we have a limit on the total number of leaves that can be traversed. We decompose the ϕ^{-1}-graph into non-overlapping paths, i.e., that no pair of paths of the decomposition shares an edge of the graph, in order to obtain a set of trees. We can build one ϕ^{-1}-tree for each path in the decomposition and use the trees collectively to compute ϕ^{-1}. Lastly, we note that we can choose any set of non-overlapping paths of the ϕ^{-1}-graph. In our implementation, we omit paths that are too short to observe speedups gained by a fast-forward query. Therefore, we deal with nodes not present in the ϕ^{-1}-forest separately.

Given the set of trees (i.e., the forest) built on the non-overlapping paths, we can use them to speed up the random access computation as follows. Let i be a position in the suffix array. We first compute the position j of the end of a BWT run immediately preceding i. The number of iterations of ϕ^{-1} to be applied to $\mathrm{SA}[j]$ are $d = i - j$. We start from $s = \mathrm{SA}[j]$, and find the node v in the ϕ^{-1}-graph corresponding to $\mathrm{pred}_{\mathcal{E}}(s)$. Subsequently, we set $c_0 = s - \mathrm{pred}_{\mathcal{E}}(s)$. Next, we check if v is stored in ϕ^{-1}-forest.

- If v is stored in a tree of ϕ^{-1}-forest (Line 8 in Algo. 2), we perform a bounded fast-forward query on this tree, starting from the node v, with initial cost c_0, and a limit to the total number of leaves to be traversed to be d. The bounded fast-forward query will return the index i_k in the graph of the reached leaf, the final cost c, and the number of traversed edges t. Hence, we compute $\mathrm{SA}[i - d + t] = \mathcal{E}[i_k] + c$, and update the remaining steps d to $d - t$.
- Otherwise (Line 12 in Algo. 2), we apply the standard computation of ϕ^{-1} to obtain $\mathrm{SA}[i - d + 1] = \mathrm{pred}_{\mathcal{E}}(s) + c_0$. Subsequently, we decrement the remaining steps d by one.

We iterate this procedure until no further steps are required to be computed, i.e., until we obtain $d = 0$. We summarize this procedure in Algo. 2. For simplicity, we assume there that we are not in the first run. Otherwise, we set $j = n = \mathcal{E}[r]$ and write $d \leftarrow i - j + n \mod n$. In ϕ^{-1}-graph, $\mathcal{E}[r]$ is always represented by a node having no outgoing edge. Therefore, we initially run into the **else** branch in Line 12.

Theorem 1. *Given a string $S[1..n]$, the r-index of S augmented by ϕ^{-1}-forest, and an index $1 \leq i \leq n$, we can compute $\mathrm{SA}[i]$ in $\mathcal{O}((i-j)(\log\log_w(n/r) + \log r))$ time, where j is the position of the end of a run preceding i and w is the machine word size, with $\mathcal{O}(r)$ additional space to the r-index.*

Algorithm 2 SA access via ϕ^{-1}-forest

1: **function** SA ACCESS(SA, i)
2: Let j be the end of a BWT run preceding i.
3: Let $d \leftarrow i - j$ and let v be the node corresponding to $SA[j]$.
4: $s \leftarrow \mathrm{SA}[j]$.
5: **while** $d > 0$ **do**
6: Let v be the node in G corresponding to $\mathrm{pred}_{\mathcal{E}}(s)$.
7: Let $c_0 = s - \mathrm{pred}_{\mathcal{E}}(s)$.
8: **if** v in one of ϕ^{-1}-forest **then**
9: Let $\mathcal{T}$ be the tree containing v.
10: $(i_k, c, t) \leftarrow$ BOUNDED FAST-FORWARD($\mathcal{T}$, v, c_0, d).
11: $s \leftarrow \mathcal{E}[i_k] + c$.
12: **else**
13: $s \leftarrow \mathrm{pred}_{\mathcal{E}}(s) + c_0$.
14: $t \leftarrow 1$.
15: $d \leftarrow d - t$.
16: **return** s

Proof. First, we note that our SA access only requires, in addition to the r-index, ϕ^{-1}-forest since all other components (including predecessor access) are part of the r-index. There exists at most one node for each SA entry in $\mathcal{E}$ in ϕ^{-1}-forest, and each node has at most one outgoing edge. Hence, it follows that ϕ^{-1}-forest requires $\mathcal{O}(r)$-space since the size of $\mathcal{E}$ is at most r. For the query time, in the case that no node is stored in any ϕ^{-1}-tree, we preform one predecessor query for each SA value between j and i, which takes $\mathcal{O}((i-j)\log\log_w(n/r))$ time. In the case that a tree is used, one bounded fast-forward query can be performed in $\mathcal{O}(\log r)$ time. Hence, we have $\mathcal{O}((i-j)(\log\log_w(n/r) + \log r))$ time. □

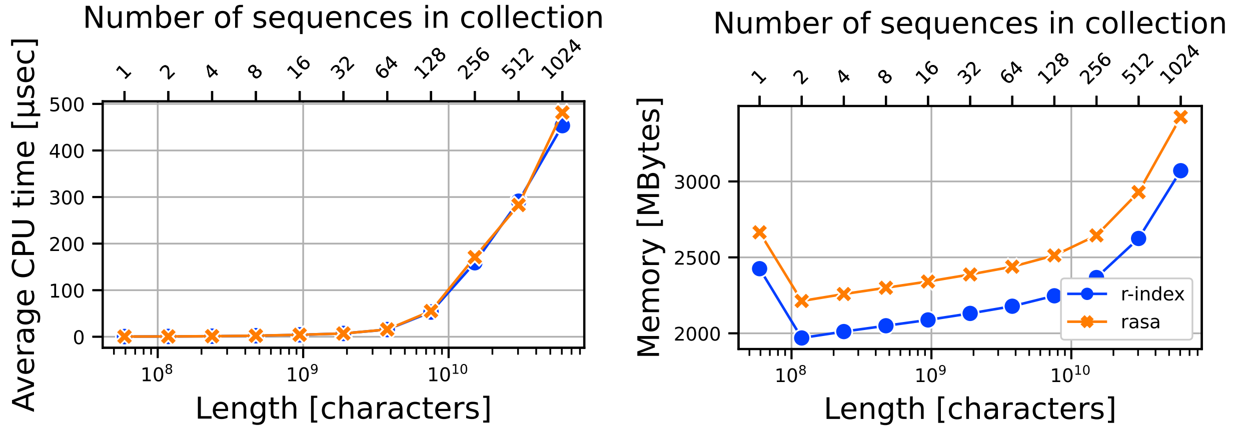

Fig. 3: Illustration of the SA access time (left) and space (right) of the `r-index` and `rasa` on increasingly larger numbers of Chromosome 19 sequences.

4 Experiments

Experimental details. We evaluated the performance on two different datasets. First, we compared the methods using Chromosome 19 sequences from the 1000 Genomes Project [17]. From this project, we created

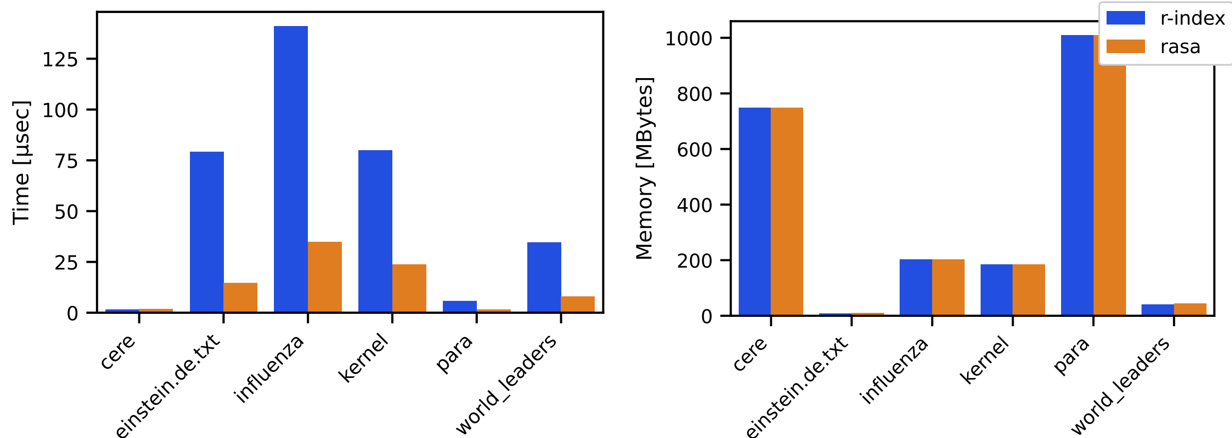

Fig. 4: Illustration of the SA access time (left) and space (right) of the `r-index` and `rasa` on Pizza&Chili repetitive corpus. We sampled 100,000 suffix array index positions between 1 and n at random and calculated the mean CPU time to perform an SA access.

datasets consisting of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 sequences, which we call `chr.`x in the following, where x is the number of assigned individuals. Next, we compared the methods using the Pizza&Chili repetitive corpus [1]. We performed all experiments on an AMD EPYC 75F3 32-core processor running at 2.95GHz with 512 GB of RAM with 64-bit Linux. Time was measured using `std::chrono::system_clock` from the C++ standard library.

Competing methods. We augmented the r-index implementation of Rossi et al. [16] with our data structure, which we refer to as `rasa`. We compared `rasa` to the standard ϕ implementation in the r-index of Rossi et al. We refer to this as `r-index`. We note that the sr-index of Cobas et al. [5] is up to 6x times smaller but showed no significant difference in the SA access time than the `r-index` since it uses the same ϕ implementation. The block-tree CSA implementation of Cáceres and Navarro [4] and RLCSA of Mäkinen et al. [11] (`rlcsa`) used standard backward search. We wanted to compare against RLZCSA [14] but no implementation is available, even upon request.

Random access. In order to evaluate our implementation of ϕ^{-1}, we randomly selected 100,000 SA entries and performed the SA random access 5 time for each entry. Figures 3 and 4 illustrate the average query times. On the Chromosome 19 datasets, the query times were comparable as they differed by at most half a microsecond. On Pizza&Chili the `r-index` the results were more pronounced as the `rasa` was between 3x and 6x times faster on all datasets except for `cere` where there was negligible difference in the running time of the methods. On all datasets, `rasa` required more memory, which is expected since it required the addition of the ϕ^{-1}-forest. However, even on the largest datasets, the additional memory was less than 3 GB.

5 Conclusion

In this paper, we introduced the concept of ϕ^{-1}-forest and demonstrated how it can be used to access the SA. Again, we note that it can be implemented analogously for ϕ. To the best of our knowledge, this is the third data structure for ϕ/ϕ^{-1} since the development of the r-index—with Nishimoto and Tabei [13] and sr-index [5] being the other two data structures. And although Brown et al. [2] implemented the LF data structure of Nishimoto and Tabei, the ϕ data structure of Nishimoto and Tabei has yet to be

implemented. Our ϕ^{-1}-forest is competitive to the standard SA access of the r-index in practice, and also provides a graphical representation that we believe can be exploited to gain further insight into decreasing the theoretical time to access an entry of the SA if we store only sampled values.

Acknowledgments. DK was supported by JSPS KAKENHI (Grant No. JP21K17701, JP21H05847, and JP22H03551). CB and MR were supported by NIH NHGRI (R01HG011392) and NSF EAGER (Grant No. 2118251). CB and HP were funded by NSF SCH:INT (Grant No. 2013998).

References

1. Pizza & Chili Repetitive Corpus. `http://pizzachili.dcc.uchile.cl/repcorpus.html`. Accessed June 2022.
2. N. K. Brown, T. Gagie, and M. Rossi. RLBWT Tricks. In C. Schulz and B. Uçar, editors, *20th International Symposium on Experimental Algorithms (SEA 2022)*, volume 233 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
3. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
4. M. Cáceres and G. Navarro. Faster repetition-aware compressed suffix trees based on block trees. *Information and Computation*, page 104749, 2021.
5. D. Cobas, T. Gagie, and G. Navarro. A Fast and Small Subsampled R-Index. In *32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
6. T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018.
7. T. Gagie, G. Navarro, and N. Prezza. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *J. ACM*, 67(1):2:1–2:54, 2020.
8. J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted Longest-Common-Prefix Array. In *Combinatorial Pattern Matching*, pages 181–192. Springer Berlin Heidelberg, 2009.
9. B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):1–10, 2009.
10. H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.
11. V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
12. U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
13. T. Nishimoto and Y. Tabei. Optimal-Time Queries on BWT-Runs Compressed Indexes. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming, (ICALP 2021)*, volume 198 of *LIPIcs*, pages 101:1–101:15, 2021.
14. S. J. Puglisi and B. Zhukova. Relative Lempel-Ziv Compression of Suffix Arrays. In *Proceedings of the 27th International Symposium on String Processing and Information Retrieval (SPIRE 2020)*, volume 12303 of *LNCS*, pages 89–96, Cham, 2020. Springer.
15. S. J. Puglisi and B. Zhukova. Smaller RLZ-Compressed Suffix Arrays. In *Proceedings of the 31st Data Compression Conference, (DCC 2021)*, pages 213–222. IEEE, 2021.
16. M. Rossi, M. Oliva, B. Langmead, T. Gagie, and C. Boucher. MONI: A Pangenomic Index for Finding Maximal Exact Matches. *Journal of Computational Biology*, 29(2):169–187, 2022.
17. The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015.