Compressed Orthogonal Search on Suffix Arrays with Applications to Range LCP

Kotaro Matsuda, Kunihiko Sadakane, Tatiana Starikovskaya, Masakazu Tateshita

Compressed Orthogonal Search on Suffix Arrays with Applications to Range LCP

Kotaro Matsuda
Graduate School of Information Science and Technology, The University of Tokyo, Japan
kotaro_matsuda@me2.mist.i.u-tokyo.ac.jp

Kunihiko Sadakane
Graduate School of Information Science and Technology, The University of Tokyo, Japan
sada@mist.i.u-tokyo.ac.jp

Tatiana Starikovskaya
DIENS, École normale supérieure, PSL Research University, Paris, France
tat.starikovskaya@gmail.com

Masakazu Tateshita
Graduate School of Information Science and Technology, The University of Tokyo, Japan
masakazu_tateshita@me2.mist.i.u-tokyo.ac.jp

Abstract

We propose a space-efficient data structure for orthogonal range search on suffix arrays. For general two-dimensional orthogonal range search problem on a set of n points, there exists an $n \log n(1+\mathrm{o}(1))$-bit data structure supporting $\mathrm{O}(\log n)$-time counting queries [Mäkinen, Navarro 2007]. The space matches the information-theoretic lower bound. However, if we focus on a point set representing a suffix array, there is a chance to obtain a space efficient data structure. We answer this question affirmatively. Namely, we propose a data structure for orthogonal range search on suffix arrays which uses $\mathrm{O}(\frac{1}{\varepsilon}n(H_0+1))$ bits where H_0 is the order-0 entropy of the string and answers a counting query in $\mathrm{O}(n^{\varepsilon})$ time for any constant $\varepsilon > 0$. As an application, we give an $\mathrm{O}(\frac{1}{\varepsilon}n(H_0+1))$-bit data structure for the range LCP problem.

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases Orthogonal Range Search, Succinct Data Structure, Suffix Array

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.23

Funding *Kunihiko Sadakane*: Supported by JST CREST Grant Number JPMJCR1402, Japan.

Acknowledgements The authors would like to thank anonymous referees for helpful comments.

1 Introduction

In this paper we consider the problem of orthogonal range search on suffix arrays (ORS on SA). The problem is defined as follows. We are given the suffix array $\mathsf{SA}[1,n]$ of a string T of length n, which is a data structure for string matching [12]. Then we construct a two-dimensional point set $P = \{p_i = (i, \mathsf{SA}[i]) \mid i = 1, \ldots, n\}$. We consider two types of queries on P: a reporting query and a counting query. Given a query region $Q = [x_1, x_2] \times [y_1, y_2]$, the reporting query must output $Q \cap P$, and the counting query $|Q \cap P|$.

The problem is a special case of the general two-dimensional range search problem for which there exists $\mathrm{O}(\log n)$ time solutions for the counting query using $\mathrm{O}(n \log n)$-bit space [4] or $n \log n(1 + \mathrm{o}(1))$-bit[1] space [11]. However there are no data structures using the fact that the point set represents a suffix array to reduce the data structure size.

[1] Throughout the paper $\log n$ denotes $\log_2 n$.

The orthogonal range search problem on suffix arrays has many applications in string algorithms. A direct application is the position-restricted substring search problem [11]. Given a pattern $P[1,m]$ and two integers $1 \leq \ell \leq r \leq n$, count all the occurrences of P in $T[l,r]$ or locate them. This corresponds to the counting and reporting problems on the point set P for the suffix array. Other problems are the interval LCP problem introduced by Cormode and Muthukrishnan [5] and the range LCP problem introduced by Amir et al. [2]. For a string $T[1,n]$, let $lcp(i,j)$ denote the length of the longest common prefix between $T[i,n]$ and $T[j,n]$. An interval LCP query $ilcp(i,\alpha,\beta)$ takes three integers $i,\alpha,\beta \in [1,n]$ and must return $\max_{j\in[\alpha,\beta]} lcp(i,j)$. A range LCP query $rlcp(\alpha,\beta)$ receives two integers $\alpha,\beta \in [1,n]$ and must return $\max_{i,j\in[\alpha,\beta]} lcp(i,j)$.

Although the orthogonal range search problem on suffix arrays plays an important role in string algorithms, a bottleneck is its space usage. For a string of length n on an alphabet of size σ, its suffix array uses $n\log n$ bits [12]. A data structure for orthogonal range search uses $n\log n + \mathrm{o}(n\log n)$ bits, which can be used solely without the standard representation of the suffix array. If $\sigma < n$, this can be much more than the $n\log\sigma$ bits of space required to store the string itself.

It seems $n\log n$ bits are necessary for storing suffix arrays or ORS data structures because it can represent a permutation of $\{1,2,\ldots,n\}$ which requires $\Omega(n\log n)$ bits to represent. For suffix arrays, however, there are data structures for storing them in $\mathrm{O}(n\log\sigma)$ bits [8] or $\mathrm{O}(n(H_0(T)+1))$ bits [18] where $H_0(T)$ is the order-0 entropy of T. This is not surprising, if we do not consider query time, because the suffix arrays is computed from the string that can be compressed in $\mathrm{O}(n(H_0(T)+1))$ bits. This should hold also for ORS data structures on suffix arrays. However there are no such data structures for ORS which have $\mathrm{o}(n)$ query time.

In this paper, we propose a space-efficient data structure for orthogonal range search on suffix arrays. The main result is as follows.

▶ **Theorem 1.** *For a string T of length n, consider orthogonal range search on the suffix array of T. For any constant $\varepsilon > 0$, there exists a data structure using $\mathrm{O}(\frac{1}{\varepsilon}\cdot n(H_0(T)+1))$ bits which supports a counting query in $\mathrm{O}(n^{\varepsilon})$ time and a reporting query in $\mathrm{O}((occ+1)\cdot n^{\varepsilon})$ time.*

This is the first data structure to achieve linear ($\mathrm{O}(n\log\sigma)$-bit) space and sub-linear query time.

As an application, we give space-efficient solutions for the interval LCP and the range LCP problems. For the interval LCP problem, [5, 9] introduced two different data structures that use $\mathrm{O}(n\log n)$ bits of space and have query time $\mathrm{O}(\log^{1+\varepsilon} n)$, where $\varepsilon > 0$ is an arbitrary constant (in fact, both works show a series of data structures with different space-time trade-offs, we only give the data structures with lowest space requirements here). In this work, we show a data structure that requires $\mathrm{O}(\frac{1}{\varepsilon}n(H_0(T)+1))$ bits of space and maintains interval LCP queries in $\mathrm{O}(n^{\varepsilon})$ time.

As for the range LCP queries, Amir et al. [2] gave two data structures; one uses $\mathrm{O}(n\log^{2+\varepsilon} n)$ bits with query time $\mathrm{O}(\log\log n)$ and the other uses $\mathrm{O}(n\log n)$ bits with query time $\mathrm{O}(\delta\log\log n)$, where $\delta = \beta-\alpha+1$ is the length of the range. Patil et al. [15] gave a data structure of $\mathrm{O}(n\log n)$ bits with query time $\mathrm{O}(\sqrt{\delta}\log^{\varepsilon}\delta)$. Abedin et al. [1] gave a data structure of $\mathrm{O}(n\log n)$ bits with query time $\mathrm{O}(\log^{1+\varepsilon} n)$. In addition, Amir, Lewenstein, and Thankanchan [3] considered range LCP queries with a bounded number of mismatches.

In this work, we develop a new data structure for range LCP queries that requires $\mathrm{O}(\frac{1}{\varepsilon}n(H_0(T)+1))$ bits of space and achieves $\mathrm{O}(n^{\varepsilon})$ query time.

2 Preliminaries

2.1 Succinct data structures

A succinct data structure is a data structure whose size asymptotically achieves the information-theoretic lower bound for representing the data. In this paper, we use succinct data structures for bit-vectors. A bit-vector is a string $B[1,n]$ on the binary alphabet $\{0,1\}$. Its information-theoretic lower bound is n bits, and there exists a succinct data structure using $n+\mathrm{o}(n)$ bits supporting the following operations in constant time [16]:

- $rank_c(B,i)$: returns the number of c's in $B[1,i]$ $(c=0,1)$,
- $select_c(B,j)$: returns the position of the j-th c from the beginning in B $(c=0,1)$.

2.2 Suffix arrays

Consider a string $T[1,n]$ of length n on an alphabet $\mathcal{A}$ of size σ. We add the unique terminator \$ at the end of the string, that is, $T[n+1]=\$$. We assume the terminator is alphabetically smaller than any letter in $\mathcal{A}$. The suffix array of T stores lexicographic order of suffixes of T. Let $\mathsf{SA}[0,n]$ be the suffix array of T. Then $\mathsf{SA}[i]=j$ means that the lexicographically i-th smallest suffix of T is the suffix $T[j,n+1]$ starting at position j of T. It holds $\mathsf{SA}[0]=n+1$ and for $i=1,\ldots,n$, $1\le \mathsf{SA}[i]\le n$ and the values are a permutation of $\{1,2,\ldots,n\}$. A pattern search for pattern length m can be done in $\mathrm{O}(m\log n)$ using the string T and its suffix array SA. The required space is $n\log\sigma$ bits for T and $n\log n$ bits for SA [12].

2.3 Compressed suffix arrays

Compressed suffix arrays [8] are data structures for compressing suffix arrays from $n\log n$ bits to $\mathrm{O}(n\log\sigma)$ bits. This is further compressed into $\mathrm{O}(n(H_0(T)+1))$ bits [18]. There are several variants of the compressed suffix arrays and the most basic one takes $\mathrm{O}(\log n)$ time to compute an entry $\mathsf{SA}[i]$ of the suffix array.

Instead of SA, compressed suffix arrays stores the Ψ function defined as follows:

$$\Psi[i]=\begin{cases}\mathsf{SA}^{-1}[\mathsf{SA}[i]+1], & \text{if } \mathsf{SA}[i]\le n;\\ \mathsf{SA}^{-1}[1], & \text{otherwise.}\end{cases}$$

Figure 1 shows an example of the suffix array and the compressed suffix array. An important property of the Ψ function is that it is piece-wise monotone.

▶ **Lemma 2** (Prop. 4.1 in [18]). *For a string of length n on an alphabet of size σ, consider its suffix array SA and Ψ. For any $1\le i<j\le n$, if $T[\mathsf{SA}[i]]=T[\mathsf{SA}[j]]$, it holds $\Psi[i]<\Psi[j]$.*

Proof. For any $1\le i\le n$, $\Psi[i]=\mathsf{SA}^{-1}[\mathsf{SA}[i]+1]$ is the lexicographic order of the suffix $T[\mathsf{SA}[i]+1,n+1]$, which is obtained by removing the first character of the suffix $T[\mathsf{SA}[i],n+1]$. For any $1\le i<j\le n$, if $T[\mathsf{SA}[i]]=T[\mathsf{SA}[j]]$, the suffixes $T[\mathsf{SA}[i],n+1]$ and $T[\mathsf{SA}[j],n+1]$ have the same first character and their relative order is determined by the suffixes made by removing the first characters. This means $\Psi[i]=\mathsf{SA}^{-1}[\mathsf{SA}[i]+1]<\mathsf{SA}^{-1}[\mathsf{SA}[j]+1]=\Psi[j]$. ◀

From this property, the Ψ function consists of at most σ increasing sequences and it can be compressed into $\mathrm{O}(n(H_0(T)+1))$ bits, and an entry of the suffix array can be computed from Ψ in $\mathrm{O}(\log n)$ time. An entry of the inverse suffix array $\mathsf{ISA}[j]=\mathsf{SA}^{-1}[j]$ can be also computed in $\mathrm{O}(\log n)$ time. For more details, see [18].

	Ψ	SA	$T[\mathsf{SA}[i], n+1]$
0	3	12	\$
1	0	11	*a*\$
2	6	8	*abra*\$
3	7	1	*abracadabra*\$
4	8	4	*acadabra*\$
5	9	6	*adabra*\$
6	10	9	*bra*\$
7	11	2	*bracadabra*\$
8	5	5	*cadabra*\$
9	2	7	*dabra*\$
10	1	10	*ra*\$
11	4	3	*racadabra*\$

	Ψ_0	SA_0	$T[SA_0[i], n+1]$
0	3	1	*abraca dabra*\$
1	4	4	*aca dabra*\$
2	5	6	*a dabra*\$
3	6	2	*braca dabra*\$
4	2	5	*ca dabra*\$
5	0	7	*dabra*\$
6	1	3	*raca dabra*\$

	Ψ'_0	SA'_0	$T[SA'_0[i], n_1+1]$
0	3	7	\$
1	4	1	*abraca*\$
2	5	4	*aca*\$
3	0	6	*a*\$
4	6	2	*braca*\$
5	3	5	*ca*\$
6	2	3	*raca*\$

Figure 1 An example of the suffix array and its sub-array.

2.4 Suffix trees and compressed suffix trees

Suffix tree [21] is a data structure for storing all the suffixes of a string T by a tree structure. Leaves of the suffix tree have one-to-one correspondence with all the suffixes. An internal node v of the suffix tree corresponds to a substring s of T so that the suffixes corresponding to the leaves in the subtree rooted at v share the same prefix s. The string depth of v is defined as the length of s.

Compressed suffix tree [19] reduces the space of the suffix tree from $\mathrm{O}(n \log n)$ bits to $6n + \mathrm{o}(n) + \mathrm{SIZE}_{SA}(n, \sigma)$ bits where $\mathrm{SIZE}_{SA}(n, \sigma)$ is the size of a data structure for the suffix array. Let $\mathrm{TIME}_{SA}(n, \sigma)$ denote the time to compute an entry of the suffix array or the inverse suffix array using a compressed suffix array. Then the string depth of a node is computed in $\mathrm{O}(\mathrm{TIME}_{SA}(n, \sigma))$ time. Also, given two positions i, j in the string, $lcp(i, j)$ is computed in $\mathrm{O}(\mathrm{TIME}_{SA}(n, \sigma))$ time. Note that there is a data structure [18] satisfying $\mathrm{SIZE}_{SA}(n, \sigma) = \mathrm{O}(n(H_0(T) + 1))$ bits and $\mathrm{TIME}_{SA}(n, \sigma) = \mathrm{O}(\log n)$.

By augmenting the compressed suffix tree with a constant time level ancestor query data structure [13], we can answer weighted level ancestor queries in $\mathrm{O}(\mathrm{TIME}_{SA}(n, \sigma) \log n)$ time. A weighted level ancestor query $WLA(u, d)$ on a suffix tree is to find the nearest ancestor of a node u in the suffix tree with string depth at most d. This is solved by simply binary searching nodes on the path from u to the root using string depths of nodes as keys.

2.5 Wavelet trees and orthogonal range search

Wavelet tree [7] is a data structure for computing *rank* and *select* on strings efficiently for large alphabets. The wavelet tree of a string T of length n on an alphabet of size σ occupies $(n + \mathrm{o}(n)) \log \sigma$ bits of space and allows computing $rank_c(T, i)$ and $select_c(T, j)$ for any character c in the alphabet in $\mathrm{O}(\log \sigma)$ time. Wavelet trees can be used to answer two-dimensional orthogonal range searches in $\mathrm{O}(\log \sigma)$ time [11]: Given a two-dimensional rectangle query $Q = [x_1, x_2] \times [y_1, y_2]$, the orthogonal range reporting query must output $Q \cap P$, and the orthogonal range counting query $|Q \cap P|$.

Because in this paper we consider wavelet trees only for strings consisting of suffix array entries, we explain the data structure of the wavelet tree using suffix arrays.

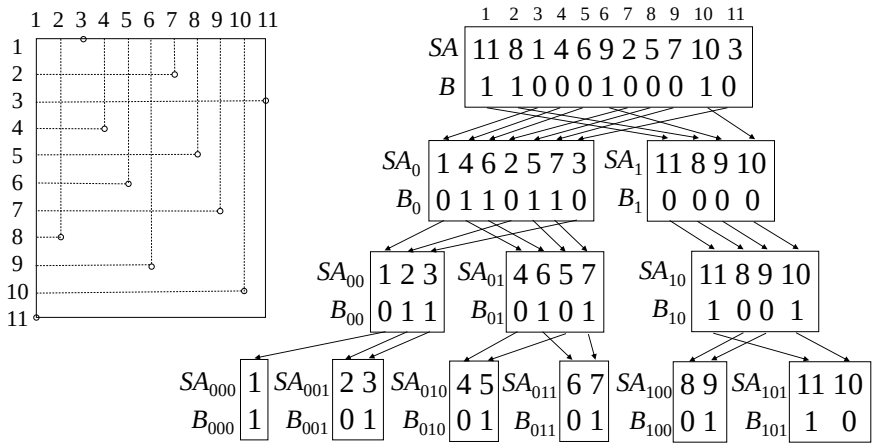

Figure 2 An example of a point set for the suffix array of Figure 1 (left), and its wavelet tree representation (right).

The wavelet tree W of the suffix array SA of length n is defined recursively. In the root node of W, we store a bit-vector $B[1,n]$, where $B[i]$ is the first bit of the binary encoding of $\mathsf{SA}[i]$ for $i = 1,\ldots,n$. We construct two arrays, SA_0 and SA_1, where SA_0 stores all the entries of SA whose first bit is 0 in the same order as in SA, and SA_1 stores the rest (all the entries of SA whose first bit is 1) in the same order as in SA. We consider that the first bits of entries of SA_0 and SA_1 are removed. The left and the right children of the root node of W store the wavelet trees for SA_0 and SA_1, respectively. We say that the root node is in level 0 and its children are in level 1, and so on. Let B_0 and B_1 denote the bit-vectors in the left child and the right child, respectively. Then the length of B_0 is $rank_0(B,n)$ and if $B[i] = 0$, $\mathsf{SA}[i]$ corresponds to $SA_0[rank_0(B,i)]$. Similarly, the length of B_1 is $rank_1(B,n) = n - rank_0(B,n)$ and if $B[i] = 1$, $\mathsf{SA}[i]$ corresponds to $SA_1[rank_1(B,i)]$. We add the auxiliary data structures for computing *rank* and *select* on the bit-vectors.

3 Compressed Orthogonal Range Search on Suffix Arrays

In this section, we prove Theorem 1 and give an $\mathrm{O}(n(H_0(T)+1))$-bit space implementation of the orthogonal range search queries on the point set $P = \{p_i = (i, \mathsf{SA}[i]) \mid i = 1,\ldots,n\}$. The wavelet tree [11] on SA occupies $n\log n + \mathrm{o}(n\log n)$ bits of space and maintains the orthogonal range queries in $\mathrm{O}(\log n)$ time. We will show that for this special case of P one can achieve $\mathrm{O}(n(H_0(T)+1))$ space complexity (at an expense of higher query time).

If we store the bit-vectors of the nodes of the wavelet tree explicitly, we need $\mathrm{O}(n\log n)$ bits. However, each bit in the bit-vectors is some bit of an entry of the suffix array, and the suffix array can be represented in $\mathrm{O}(n(H_0(T)+1))$ bits. We will use this fact to develop a data structure for orthogonal range queries by imitating the wavelet tree and decoding the required information on the fly from the compressed suffix array of the string.

3.1 Orthogonal range search

In the orthogonal range search algorithm using wavelet trees, we need to compute $rank(B_v, i)$ on the bit-vector B_v of node v of the wavelet tree. To compute it using the compressed suffix array, we need the relation between bits of the bit-vector and entries of the suffix array.

We divide B into blocks of length Δ, where Δ is a parameter to be determined later. Let $B^j = B[j\Delta+1,(j+1)\Delta]$ be the j-th block ($j = 0,1,\ldots,n/\Delta$). For each block j, we store $B^j.rank_0$: the number of 0's in the blocks to the left of B^j.

Similarly, for the bit-vector B_v of node v, we create $\lceil \frac{n}{\Delta} \rceil$ many blocks B_v^j. The block B_v^j contains the entries corresponding to the entries of SA such that they belong to B^j and their binary representation starts with v. For the j-th block B_v^j, we store

- $B_v^j.rank_0$: the number of 0's in the blocks to the left of B_v^j,
- $B_v^j.blocknum$: the total number of bits in the blocks to the left of B_v^j.

To compute $rank_0(B_v, i)$, we first find the block j of the bit-vector in the root node that contains the i-th bit of B_v, extract the entries of the compressed suffix array for the block of length Δ, compute the number of 0's in the block up to position i, and add it to the number of 0's before the block in B_v, which is stored in $B_v^j.rank_0$.

However, this approach requires even more space than the wavelet tree: at depth $\log n$, there are n nodes and therefore $\lceil \frac{n}{\Delta} \rceil \cdot n$ blocks. To overcome this, we stop the recursion for blocks at every $k = \lceil \varepsilon \log n \rceil$ steps, and restart the data structure construction. Details are as follows.

Consider the nodes of the wavelet tree at depth dk ($d = 0,1,\ldots,\frac{1}{\varepsilon}-1$). There are 2^{dk} such nodes, and for each such node v', we divide the bit-vector $B_{v'}$ into $\lceil \frac{n}{\Delta \cdot 2^{dk}} \rceil$ many blocks $B_{v'}^j$ ($j = 0,\ldots,\lfloor \frac{n}{\Delta \cdot 2^{dk}} \rfloor$) of length Δ each. Similarly to the above data structure, we store, for each j, $B_{v'}^j.rank_0$: the number of 0's in the blocks to the left of block $B_{v'}^j$.

Then, each bit-vector $B_{v''}$ of a node v'' at depth $dk+1$ to $(d+1)k$ is divided into $\lceil \frac{n}{\Delta \cdot 2^{dk}} \rceil$ many blocks $B_{v''}^j$, corresponding to the blocks of a bit-vector in a node at depth dk which is the ancestor of v'' with that level. For the j-th block $B_{v''}^j$, we store

- $B_{v''}^j.rank_0$: the number of 0's in the blocks to the left of $B_{v''}^j$,
- $B_{v''}^j.blocknum$: the total number of bits in the blocks to the left of $B_{v''}^j$.

The number of blocks between depths $dk{+}1$ and $(d{+}1)k$ is $\lceil \frac{n}{\Delta \cdot 2^{dk}} \rceil \cdot 2^k \cdot 2^{dk} = 2^k \cdot \lceil \frac{n}{\Delta} \rceil = n^{\varepsilon} \cdot \lceil \frac{n}{\Delta} \rceil$. Therefore, the total space is $O(\frac{1}{\varepsilon} \cdot n^{\varepsilon} \frac{n}{\Delta} \log n)$ bits.

At depth dk of the wavelet tree, there are 2^{dk} many bit-vectors $B_{00\ldots00}$, $B_{00\ldots01}$, …, $B_{11\ldots11}$. Each bit-vector corresponds to a sub-array SA_v of SA such that the first dk bits of the binary encodings of the suffix array entries are equal to v. That is, the bit-vector $B_{00\ldots00}$ corresponds to the sub-array $SA_{00\ldots00}$ containing entries in the range $[1, \frac{n}{2^{dk}}]$, $B_{00\ldots01}$ to $SA_{00\ldots01}$ containing entries in $[\frac{n}{2^{dk}}+1, \frac{2n}{2^{dk}}]$, and $B_{11\ldots11}$ to $SA_{11\ldots11}$ containing entries in $[\frac{(2^{dk}-1)n}{2^{dk}}+1, n]$.

▶ **Lemma 3.** *The sub-array SA_v, which consists of the entries $SA[\ell(v-1)+1, \ell v]$ where $\ell = \frac{n}{2^{dk}}$, contains positions in the substring $T_v = T[\ell(v-1)+1, \ell v]$, and it can be compressed in $O(\ell(H_0(T_v)+1))$ bits. Each entry of SA_v can be computed in $O(\log \ell)$ time.*

Proof. SA_v stores positions of suffixes between $\ell(v-1)+1$ and ℓv. From the construction of the wavelet tree, SA_v stores positions of T_v. We insert one entry $\ell v + 1$ to the sub-array, whose position in the sub-array is determined by the lexicographic order of the corresponding suffixes in the entire string. Let p be this position. We subtract $\ell(v-1)+1$ from each entry in SA_v so that they become integers from 0 to ℓ to obtain an array SA'_v. (See Figure 1 for an example.) For any $0 \le i \le v+1$ except for p, we define $\Psi'_v[i] = {SA'_v}^{-1}[SA'_v[i]+1]$. By Lemma 2, Ψ'_v consists of at most σ increasing sequences and the values are in $[0,\ell]$. Then we can compress Ψ'_v in $O(\ell(H_0(T_v)+1))$ bits, and each entry of $SA'_v[i]$ can be computed in $O(\log \ell)$ time (Theorem 4.1 in [18]). ◀

Algorithm 1 *calcblock*$(v, level, k)$: Given a node v with depth $level$, find the node v' that is the nearest ancestor of v whose depth is a multiple of k.

```
1: v' = v
2: for i = 1, ..., level − k do
3:     v' = ⌊(v'−1)/2⌋
4: return v'
```

Algorithm 2 *calcblockrank*$(v, level, j, c, k)$: returns the $rank_0$ value at $B_v[c]$ in the bit-vector B_v by extracting Δ entries of the compressed suffix array in the block that contains $B_v[c]$.

```
1: blockvalue = c − B_v^j.blocknum
2: v' = calcblock(v, level, k)
3: rank = 0
4: i = 1
5: cnt = 0
6: while cnt < blockvalue do
7:     if the highest (level − k · ⌊level/k⌋) bits of SA_{v'}[j · Δ + i] are equal to those of v then
8:         cnt = cnt + 1
9:         if the (level − k · ⌊level/k⌋ + 1)-st bit of SA_{v'}[j · Δ + i] is 0 then
10:            rank = rank + 1
11:    i = i + 1
12: return rank
```

For each node v of depth dk of the wavelet tree, we represent SA_v as in Lemma 3. Therefore, the total space for depth dk is $\sum_v \mathrm{O}(\ell(H_0(T_v) + 1))$ bits. From concavity of logarithm, this is upper bounded by $\mathrm{O}(n(H_0(T) + 1))$ bits. Then the total for all levels which are multiple of k is $\mathrm{O}(\frac{1}{\varepsilon} \cdot n(H_0(T) + 1))$ bits. Levels that are not multiples of k require $\mathrm{O}(\frac{1}{\varepsilon} \cdot n^{\varepsilon} \frac{n}{\Delta} \log n)$ bits of space.

Using this data structure for orthogonal range search, the counting query is solved by Algorithms 1–3. The *rank* operations take $\mathrm{O}(\Delta \cdot \log n)$ time for nodes of depths which are multiples of k, and $\mathrm{O}(\Delta)$ time for other nodes. Therefore, the total query time is $\mathrm{O}(\Delta \cdot \log n \cdot \frac{1}{\varepsilon} + \Delta \cdot \log n) = \mathrm{O}(\Delta \cdot \log n)$. The reporting query is solved analogously, and for output of size occ takes $\mathrm{O}(occ \cdot \Delta \cdot \log n)$ time. By letting $\Delta = n^{\varepsilon} \log n$, we obtain a data structure of $\mathrm{O}(\frac{1}{\varepsilon} \cdot n(H_0(T) + 1))$ bits and $\mathrm{O}(n^{\varepsilon} \log^2 n)$ query time. To improve the query time to $\mathrm{O}(n^{\varepsilon})$, we use another parameter $\varepsilon' < \varepsilon$ such that $n^{\varepsilon'} \log^2 n = \mathrm{O}(n^{\varepsilon})$ and $\frac{1}{\varepsilon'} = \mathrm{O}(\frac{1}{\varepsilon})$. Then we obtain Theorem 1.

3.2 Range successor/predecessor

As a simple extension, we consider 2-dimensional range successor and predecessor queries [14] defined as follows.

▶ **Definition 4.** *Let P be a set of n points on the $[1, n] \times [1, n]$ grid. The two-dimensional range successor and predecessor queries are defined as:*

$$
\begin{aligned}
ORS_{xSucc}([x, +\infty], [y, y']) &= \mathrm{argmin}_i\{(i, j) \in P \cap [x, +\infty] \times [y, y']\},\\
ORS_{xPred}([-\infty, x'], [y, y']) &= \mathrm{argmax}_i\{(i, j) \in P \cap [-\infty, x'] \times [y, y']\},\\
ORS_{ySucc}([x, x'], [y, +\infty]) &= \mathrm{argmin}_j\{(i, j) \in P \cap [x, x'] \times [y, +\infty]\},\\
ORS_{yPred}([x, x'], [-\infty, y']) &= \mathrm{argmax}_j\{(i, j) \in P \cap [x, x'] \times [-\infty, y']\}.
\end{aligned}
$$

Algorithm 3 $ORS\_SA([x_1, x_2], [y_1, y_2])$: For the point set $p_i = (i, \mathsf{SA}[i])$ $(i = 1, \ldots, n)$ where SA is the suffix array of a string of length n, return the number of points $p_i \in Q = [x_1, x_2] \times [y_1, y_2]$.

return $count([x_1, x_2], [y_1, y_2], \varepsilon, 0, [0, 0], [1, n], k)$

function $count([x_1, x_2], [y_1, y_2], v, level, j_1, j_2, [a, b], k)$
$\#leftchild(v)$: returns the bit-vector of the left child of v, i.e., the bit-vector of v to which 0 is appended at the end.
$\#rightchild(v)$: returns the bit-vector of the right child of v, i.e., the bit-vector of v to which 1 is appended at the end.
if $x_1 > x_2$ **then**
  return 0
if $[a, b] \cap [y_1, y_2] = \emptyset$ **then**
  return 0
if $[a, b] \subset [y_1, y_2]$ **then**
  return $x_2 - x_1 + 1$
$k = \lceil \varepsilon \log n \rceil$
if $level \ \% \ k = 0$ **then**
  $j_1 = \lfloor \frac{x_1 - 1}{\Delta} \rfloor$
  $j_2 = \lfloor \frac{x_2}{\Delta} \rfloor$
$x_1^l = calcblockrank(v, level, j_1, x_1 - 1, k) + B_v^{j_1}.rank_0 + 1$
$x_2^l = calcblockrank(v, level, j_2, x_2, k) + B_v^{j_1}.rank_0$
$x_1^r = x_1 - x_1^l + 1$
$x_2^r = x_2 - x_2^l$
$m = \lfloor \frac{a+b}{2} \rfloor$
return $count([x_1^l, x_2^l], [y_1, y_2], leftchild(v), level + 1, j_1, j_2, [a, m], k) +$
$count([x_1^r, x_2^r], [y_1, y_2], rightchild(v), level + 1, j_1, j_2, [m + 1, b], k)$

We focus on range successor and predecessor queries on suffix arrays. In addition to the data structure in Section 3.1, we store the following.

- $B_v^j.xSucc$: the index i such that $\mathsf{SA}[i]$ attains the minimum value in blocks B_v^j to $B_v^{\lfloor \frac{n}{\Delta} \rfloor}$.
- $B_v^j.xPred$: the index i such that $\mathsf{SA}[i]$ attains the maximum value in blocks B_v^0 to B_v^j.
- $B_v^j.ySucc$: $\min\{\mathsf{SA}[x] \mid x \in [\frac{n}{\Delta} \cdot j, \frac{n}{\Delta} \cdot (j+1) - 1]$ and $\mathsf{SA}[x] \in [\ell, n]\}$ where ℓ is the leftmost index corresponding to node v
- $B_v^j.yPred$: $\max\{\mathsf{SA}[x] \mid x \in [\frac{n}{\Delta} \cdot j, \frac{n}{\Delta} \cdot (j+1) - 1]$ and $\mathsf{SA}[x] \in [0, r]\}$ where r is the rightmost index corresponding to node v

Using these data structures, we can answer $ORS_{xSucc}([x, +\infty], [y, y'])$ ($ORS_{xPred}([-\infty, x'], [y, y'])$) as follows. First, we extract $\mathsf{SA}[x, (\lfloor \frac{x}{\Delta} \rfloor + 1) \cdot \Delta - 1]$ ($\mathsf{SA}[\lfloor \frac{x'}{\Delta} \rfloor \cdot \Delta, x']$), and if there is a value in $[y, y']$, the one with the smallest (largest) index is the answer. This is done in $O(\Delta \cdot \log n)$ time. Otherwise, we perform an ORS query. During the search every time we compute $rank(B_v, c)$, if $B_v[c]$ belongs to B_v^j and the range of characters $[s, t]$ for node v satisfies $[s, t] \subset [y, y']$, we store $B_v^{j-1}.xSucc$ ($B_v^{j-1}.xPred$) as a candidate of the answer and compare it with the minimum (maximum) index of the extracted suffix array values in $[y, y']$ If $\Delta = n^\varepsilon \log n$, the time complexity is $O(\Delta \cdot \log n + \Delta \cdot \log n) = O(n^\varepsilon \log^2 n)$. This can be easily reduced to $O(n^\varepsilon)$ time. $ORS_{ySucc}([x, x'], [y, +\infty])$ ($ORS_{yPred}([x, x'], [-\infty, y'])$) is solved similarly.

We obtain the following.

▶ **Theorem 5.** *For a string T of length n, there exists an $O(\frac{1}{\varepsilon} \cdot n(H_0(T)+1))$-bit data structure supporting two-dimensional range successor/predecessor queries in $O(n^{\varepsilon})$ time.*

Keller et al. [10] showed the following.

▶ **Lemma 6** (cf. [10]). *The interval LCP queries can be reduced to two-dimensional range successor/predecessor queries.*

Proof. We reduce an interval LCP query $ilcp(p, \alpha, \beta)$ to the two-dimensional range successor/predecessor queries on a set $P = \{(i, \mathsf{SA}[i]) \mid i \in [1, n]\}$ for the string. Let $q = \mathsf{SA}^{-1}[p]$. We compute the largest lexicographic order $x < q$ such that $\mathsf{SA}[x] \in [\alpha, \beta]$ and the smallest lexicographic order $y > q$ such that $\mathsf{SA}[y] \in [\alpha, \beta]$. This is done by $x = ORS_{xPred}([-\infty, p-1], [\alpha, \beta])$ and $y = ORS_{xSucc}([p+1, +\infty], [\alpha, \beta])$. Then it holds $ilcp(p, \alpha, \beta) = \max\{lcp(p, x), lcp(p, y)\}$. ◀

From Theorem 5 and Lemma 6, we obtain:

▶ **Corollary 7.** *For a string T of length n, there exists an $O(\frac{1}{\varepsilon} \cdot n(H_0(T)+1))$-bit data structure supporting interval LCP queries in $O(n^{\varepsilon})$ time.*

4 Range LCP queries

In this section, we will show a data structure that occupies $O(\frac{1}{\varepsilon} n(H_0(T)+1))$ bits of space and supports *rlcp* queries in $O(n^{\varepsilon})$ time. We follow the outline of the data structure presented in [1], but improve the space complexity of the key components of the data structure.

▶ **Definition 8** (Bridges). *Let i and j be two distinct positions in the string T with $i < j$ and let $h = lcp(i, j)$ and $h > 0$. Then, we call the tuple (i, j, h) a bridge. Moreover, we call h its height, i its left leg and j its right leg, and $lcp(T[i, n], T[j, n])$ its label.*

The set of all bridges is denoted by $\mathcal{B}_{all}$. Next, we will define the set of special bridges $\mathcal{B}_{spe}$ via the heavy-path decomposition of the suffix tree ST of the string T.

▶ **Definition 9** (Heavy Path Decomposition [20]). *Let τ be a tree. The nodes in τ are categorised into light and heavy ones. The root node is light. Furthermore, for each node of τ, exactly one of its children is heavy and the others are light. The heavy child has the largest number of leaves in its subtree among the children (ties are broken arbitrarily). When all edges incoming to light nodes are deleted, τ is decomposed into (heavy) paths.*

Recall that there is one-to-one correspondence between the suffixes of T and the leaves of ST. By ℓ_i we denote the leaf corresponding to $T[\mathsf{ISA}[i], n]$.

▶ **Definition 10** (Special Bridges). *Let u_i, u_j in ST be the children of the lowest common ancestor of ℓ_i and ℓ_j on the path to ℓ_i, ℓ_j, respectively. A bridge $(i, j, h) \in \mathcal{B}_{all}$ is special if it satisfies at least one of two following conditions.*

1. *u_i is a light node* ***and*** *$j = \min\{x \mid (i, x, h) \in \mathcal{B}_{all}\}$*
2. *u_j is a light node* ***and*** *$i = \max\{x \mid (x, j, h) \in \mathcal{B}_{all}\}$*

The reason to introduce this definition is that we can express range LCP queries via the special bridges.

▶ **Lemma 11** (cf. [1]). *Let $\mathcal{B}_{spe}$ be the set of all special bridges. Then $|\mathcal{B}_{spe}| = O(n \log n)$ and for any α, β we have $rlcp(\alpha, \beta) = \max\{h : (i, j, h) \in \mathcal{B}_{spe}$ and $i, j \in [\alpha, \beta]\}$.*

▶ **Definition 12** (cf. [1]). *For $i, j \in \mathbb{N}$ and $h \in \mathbb{N}$, we define*

$$\mathsf{crightLeg}(i,h) = \begin{cases} \min\{x \mid (i,x,h) \in \mathcal{B}_{spe}\} \text{ if there exists a bridge } (i,\cdot,h) \in \mathcal{B}_{spe}; \\ +\infty, \text{ otherwise.} \end{cases}$$

$$\mathsf{cleftLeg}(j,h) = \begin{cases} \max\{x \mid (x,j,h) \in \mathcal{B}_{spe}\} \text{ if there exists a bridge } (\cdot,j,h) \in \mathcal{B}_{spe}; \\ -\infty, \text{ otherwise.} \end{cases}$$

▶ **Lemma 13.** *By maintaining a data structure of space $\mathrm{O}(\frac{1}{\varepsilon}n(H_0(T)+1))$ bits, we can answer $\mathsf{crightLeg}(k,h)$ and $\mathsf{cleftLeg}(k,h)$ queries in $\mathrm{O}(n^{\varepsilon})$ time.*

Proof. By [1], we can reduce computing $\mathsf{crightLeg}(k,h)$ and $\mathsf{cleftLeg}(k,h)$ to four range successor/predecessor queries on the suffix array of T and standard operations on ST. The claim follows by Theorem 1. ◀

▶ **Lemma 14** (cf. [1]). *Suppose that $(i,j,h) \in \mathcal{B}_{spe}$. Then, $\forall k \in [1, h-1]$, there exists at least one $(i+k, \cdot, h-k) \in \mathcal{B}_{spe}$ such that $\mathsf{crightLeg}(i+k, h-k) \in (i+k, j+k]$ and $(\cdot, h+k, h-k) \in \mathcal{B}_{spe}$ such that $\mathsf{crightLeg}(i+k, h-k) \in (i+k, j+k]$.*

Let S be a set of m weighted points in a $[1,n] \times [1,n]$ grid. A 2D-RMQ with input (a,b,a',b') asks to return the highest weighted point in S within the orthogonal region corresponding to $[a,b] \times [a',b']$. There is a data structure for this problem that uses $\mathrm{O}(m \log n)$ bits of space and $\mathrm{O}(\log^{1+\gamma} m)$ query time [4]. Let $\Delta = \log^2 n$ and $\mathcal{B}_{spe}^{x \pmod \Delta}$ be the set of special bridges of heights congruent to x modulo Δ. For some $\pi \in [0, \Delta - 1]$ we have $|\mathcal{B}_{spe}^{\pi \pmod \Delta}| = \mathrm{O}(n/\log n)$ as a corollary of Lemma 11 and the Pigeonhole principle. We map each special bridge $(i,j,h) \in \mathcal{B}_{spe}^{\pi}$ into a $2D$ point (i,j) with weight h and maintain the 2D-RMQ data structure over these points.

Let (α^*, β^*, h^*) be the tallest special bridge, such that both $\alpha^*, \beta^* \in [\alpha, \beta]$. We query the 2D-RMQ structure and find the tallest bridge $(i', j', h') \in \mathcal{B}_{spe}^{\pi \pmod \Delta}$, such that $i', j' \in [\alpha, \beta]$.

▷ Claim 15. If $h^* \geq \pi$, then (i', j', h') is well-defined. Furthermore, we have $\tau \leq rlcp(\alpha, \beta) \leq \tau + \Delta$, where $\tau = \max\{ilcp(p, \alpha, \beta) | p \in (\beta - \Delta, \beta]\} \cup \{h'\}$.

Proof. If $\beta^* \in (\alpha, \beta - \Delta]$, then there is $h^* \in [h', h' + \Delta)$ by Lemma 14. Else, there is $h^* = \max\{ilcp(p, \alpha, \beta) : p \in (\beta - \Delta, \beta]\}$. ◀

Below, we consider two possible cases: $h^* < \pi$ and $\pi \geq h^*$. In the second case, by Claim 15, we can find τ such that $\tau \leq rlcp(\alpha, \beta) \leq \tau + \Delta$ in $\mathrm{O}(\log^{\gamma} n + \Delta \cdot n^{\varepsilon}) = \mathrm{O}(n^{\varepsilon'})$ time for some $\varepsilon' > \varepsilon$. We will now explain how to find the true value of h^*.

4.1 Case 1: $\pi \leq h^*$

By Claim 15, in this case we know the interval $[\pi + \Delta \cdot k, \pi + \Delta \cdot (k+1))$ that contains all possible values for h^*. Therefore, to find the true value of h^* in this case it suffices to check, for each $h \in [\pi + \Delta \cdot k, \pi + \Delta \cdot (k+1)]$ if there is a special bridge (i,j,h) with legs $i, j \in [\alpha, \beta]$.

Recall that $\mathcal{B}_{spe}^{h}$ is the set of all special bridges with height h. For each $h \in [1,n]$, we maintain a separate structure that can answer whether there is a bridge of height h with legs in $[\alpha, \beta]$. For $k = 1, 2, \ldots, |\mathcal{B}_{spe}^{h}|$, let $L_h[k]$ (resp., $R_h[k]$) denote the left leg (resp., right leg) of k-th bridge among all bridges in $\mathcal{B}_{spe}^{h}$ in the ascending order of the left (resp., right) legs. Based on whether $h \equiv \pi \pmod \Delta$ or not, we have two subcases.

4.1.1 Subcase 1a: $h \equiv \pi \pmod{\Delta}$

Let $RL_h[k] = \mathsf{crightLeg}(L_h[k], h)$ for all $k = 1, 2, \ldots, |\mathcal{B}^h_{spe}|$. We maintain the y-fast trie [22] over L_h and a succinct range minimum query data structure [6, 17] over RL_h. The total space is $\mathrm{O}(|\mathcal{B}^{\pi \pmod{\Delta}}_{spe}| \log n) = \mathrm{O}(n)$ bits. We can then decide if there is a bridge (i, j, h) such that $i, j \in [\alpha, \beta]$ via the following steps:

- Find the smallest k such that $L_h[k] \geq \alpha$ using the y-fast trie;
- Find the index k corresponding to the smallest element in $RL_h[k, |\mathcal{B}^h_{spe}|]$ using a range minimum query;
- Find $\mathsf{crightLeg}(L_h[k], h)$ and report YES if it is $\leq \beta$, and report NO otherwise.

The time complexity is $\mathrm{O}(\log \log n + n^{\varepsilon}) = \mathrm{O}(n^{\varepsilon})$ by Lemma 13.

4.1.2 Subcase 1b: $h \not\equiv \pi \pmod{\Delta}$

Let q be the largest integer smaller than h that is congruent to $\pi \pmod{\Delta}$ and $z = (h - q)$. Note that for each special bridge (i, j, h), there exists at least one special bridge $(i + z, j', h - z) = (i + z, j', q)$ and $(i', j + z, h - z) = (i', j + z, q)$ (see Lemma 14).

Now, define arrays RL_h and LR_h of length $|\mathcal{B}^q_{spe}|$, such that for any $k = 1, 2, \ldots, \mathcal{B}_{spe}$, $RL_h[k] = \mathsf{crightLeg}(L_q[k] - z, h)$ and $LR_h[k] = \mathsf{cleftLeg}(R_q[k] - z, h)$. To handle Subcase 1b, we maintain y-fast tries over L_q and R_q, and a range minimum query data structure over RL_h and a range maximum query data structure over LR_h for each $h \neq \pi \pmod{\Delta}$.

▶ **Lemma 16.** *The range minimum query data structures over RL_h and the range maximum query data structures over LR_h, for all $h \neq \pi \pmod{\Delta}$, can be implemented in $\mathrm{O}(1/\varepsilon \cdot n(H_0(T) + 1))$ bits of space with query time $\mathrm{O}(n^{\varepsilon})$.*

Proof. We show how to implement the data structures for the arrays RL_h, for the arrays LR_h they can be implemented analogously.

First, we store the data structure of Lemma 13. Second, for each h, we divide the array RL_h into non-overlapping blocks of length $\log n$. For each block, we compute the minimum value in it to obtain an array RL'_h of length $|\mathcal{B}^q_{spe}|/\log n$. On top of RL'_h, we maintain a succinct range minimum query data structure. In total, the data structures occupy $\mathrm{O}(1/\varepsilon \cdot n(H_0(T) + 1) + \Delta \cdot |\mathcal{B}^{\pi \pmod{\Delta}}_{spe}| \log n) = \mathrm{O}(1/\varepsilon \cdot n(H_0(T) + 1))$ bits of space.

To answer a range minimum query on RL'_h, we first find the index of a block that contains the minimum, and then compute the value of each entry in the block using the data structure of Lemma 13. ◀

We can now decide if there is a bridge (i, j, h) such that $i, j \in [\alpha, \beta]$ via the following steps:

- Find the smallest k such that $L_q[k] - z \geq \alpha$ using the y-fast trie over L_q.
- Find the index k' corresponding to the smallest element in $RL_h[k, |\mathcal{B}^q_{spe}|]$ using a range minimum query.
- Return YES if $\mathsf{crightLeg}(L_q[k'] - z, h) \leq \beta$, otherwise continue to the next step.
- Find the largest l such that $R_q[l] - z \leq \beta$. We use the y-fast trie for R_q for this.
- Find the index l' corresponding to the largest element in $LR_h[1, l]$ using a range maximum query.
- Return YES if $\mathsf{cleftLeg}(R_q[l'] - z, h) \geq \alpha$, and return NO otherwise.

The time complexity is $\mathrm{O}(n^{\varepsilon} \operatorname{polylog} n)$, and the space complexity is $\mathrm{O}(\frac{1}{\varepsilon} n(H_0(T) + 1))$ bits.

4.2 Case 2: $h^* < \pi$

In this case, we must check, for every $0 < h < \pi$, if there is a special bridge (i, j, h), where $i, j \in [\alpha, \beta]$. We will use the same reduction as in Case 2 with $q = 1$.

Note that $\mathcal{B}^q_{spe}$ can be of size $\Theta(n)$, so we cannot store the y-fast trie for L_q and R_q, it could take $\Theta(n \log n)$ bits of space. Recall, however, that we only use the y-fast tries to answer predecessor (resp., successor) queries: given an integer x, find the smallest (resp., the largest) index k such that $L_q[k]$ (resp., $R_q[k]$) is larger or equal (resp., smaller or equal) to x.

▶ **Lemma 17.** *The predecessor (resp., successor) queries on L_q (resp., R_q) can be answered in $O(n)$ bits of space and $O(1)$ query time.*

Proof. We show a data structure for L_q, a data structure for R_q can be defined analogously. Let L'_q be the sequence obtained by encoding the number of special bridges (i, j, q) with $i = k$ for all $k (1 \leq k \leq n)$ in unary code and concatenating them in order. $|L'_q| \leq 3n$ because there are at most $2n$ special bridges with height q by the definition. Therefore, to answer a predecessor query for an integer x it suffices to answer *rank* and *select* queries on L'_q: we return $rank_1(L'_q, select_0(L'_q, x)) + 1$. Both *rank* and *select* queries can be answered in $O(n)$ bits of space and $O(1)$ time. ◀

By using the data structures above, we can use our solution for Case 1 to obtain similar complexities.

References

1 Paniz Abedin, Arnab Ganguly, Wing-Kai Hon, Yakov Nekrich, Kunihiko Sadakane, Rahul Shah, and Sharma V. Thankachan. A linear-space data structure for range-lcp queries in poly-logarithmic time. In *COCOON'18*, pages 615–625, 2018. doi:10.1007/978-3-319-94776-1_51.

2 Amihood Amir, Alberto Apostolico, Gad M. Landau, Avivit Levy, Moshe Lewenstein, and Ely Porat. Range LCP. *J. Comput. Syst. Sci.*, 80(7):1245–1253, 2014. doi:10.1016/j.jcss.2014.02.010.

3 Amihood Amir, Moshe Lewenstein, and Sharma V. Thankachan. Range LCP queries revisited. In *SPIRE'15*, pages 350–361, 2015. doi:10.1007/978-3-319-23826-5_33.

4 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988. doi:10.1137/0217026.

5 Graham Cormode and S. Muthukrishnan. Substring compression problems. In *SODA'05*, pages 321–330, 2005.

6 Johannes Fischer and Volker Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *ESCAPE'07*, pages 459–470, 2007. doi:10.1007/978-3-540-74450-4_41.

7 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *SODA'03*, page 841–850, 2003.

8 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. on Computing*, 35(2):378–407, 2005.

9 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. doi:10.1016/j.tcs.2013.10.010.

10 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. Advances in Stringology. doi:10.1016/j.tcs.2013.10.010.

11 Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007. doi:10.1016/j.tcs.2007.07.013.

12 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

13 Gonzalo Navarro and Kunihiko Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):Article No. 16, 39 pages, 2014.

14 Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In *SWAT'12*, pages 271–282, 2012.

15 Manish Patil, Rahul Shah, and Sharma V. Thankachan. Faster range LCP queries. In *SPIRE'13*, pages 263–270, 2013. doi:10.1007/978-3-319-02432-5_29.

16 Rajeev Raman, Venkatesh Raman, and Srinavasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4), 2007. doi:10.1145/1290672.1290680.

17 Kunihiko Sadakane. Space-efficient data structures for flexible text retrieval systems. In *ISAAC'02*, pages 14–24, 2002. doi:10.1007/3-540-36136-7_2.

18 Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. of Algorithms*, 48(2):294–313, 2003.

19 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

20 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.

21 Peter Weiner. Linear pattern matching algorithms. In *SWAT(FOCS)'73*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

22 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.