

Wavelet Trees Meet Suffix Trees*

Maxim Babenko¹, Paweł Gawrychowski², Tomasz Kociumaka³, and Tatiana Starikovskaya¹

¹National Research University Higher School of Economics (HSE)

mbabenko@hse.ru, tstarikovskaya@hse.ru

²Max-Planck-Institut für Informatik

gawry@cs.uni.wroc.pl

³Institute of Informatics, University of Warsaw

kociumaka@mimuw.edu.pl

Abstract

We present an improved wavelet tree construction algorithm and discuss its applications to a number of rank/select problems for integer keys and strings.

Given a string of length n over an alphabet of size $\sigma \leq n$, our method builds the wavelet tree in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time, improving upon the state-of-the-art algorithm by a factor of $\sqrt{\log n}$. As a consequence, given an array of n integers we can construct in $\mathcal{O}(n \sqrt{\log n})$ time a data structure consisting of $\mathcal{O}(n)$ machine words and capable of answering rank/select queries for the subranges of the array in $\mathcal{O}(\log n / \log \log n)$ time. This is a $\log \log n$ -factor improvement in query time compared to Chan and Pătraşcu (SODA 2010) and a $\sqrt{\log n}$ -factor improvement in construction time compared to Brodal et al. (Theor. Comput. Sci. 2011).

Next, we switch to stringological context and propose a novel notion of *wavelet suffix trees*. For a string w of length n , this data structure occupies $\mathcal{O}(n)$ words, takes $\mathcal{O}(n \sqrt{\log n})$ time to construct, and simultaneously captures the combinatorial structure of substrings of w while enabling efficient top-down traversal and binary search. In particular, with a wavelet suffix tree we are able to answer in $\mathcal{O}(\log |x|)$ time the following two natural analogues of rank/select queries for suffixes of substrings:

- 1) For substrings x and y of w (given by their endpoints) count the number of suffixes of x that are lexicographically smaller than y ;
- 2) For a substring x of w (given by its endpoints) and an integer k , find the k -th lexicographically smallest suffix of x .

We further show that wavelet suffix trees allow to compute a run-length-encoded Burrows-Wheeler transform of a substring x of w (again, given by its endpoints) in $\mathcal{O}(s \log |x|)$ time, where s denotes the length of the resulting run-length encoding. This answers a question by Cormode and Muthukrishnan (SODA 2005), who considered an analogous problem for Lempel-Ziv compression.

All our algorithms, except for the construction of wavelet suffix trees, which additionally requires $\mathcal{O}(n)$ time in expectation, are deterministic and operate in the word RAM model.

*The third author is supported by Polish budget funds for science in 2013-2017 as a research project under the ‘Diamond Grant’ program. The fourth author is partly supported by Dynasty Foundation.

1 Introduction

Let Σ be a finite ordered non-empty set which we refer to as an *alphabet*. The elements of Σ are called *characters*. characters are treated as integers in a range $[0, \sigma - 1]$; and we assume that a pair of characters can be compared in $\mathcal{O}(1)$ time. A finite ordered sequence of characters (possibly empty) is called a *string*. characters in a string are enumerated starting from 1, that is, a string w of length n consists of characters $w[1], w[2], \dots, w[n]$.

Wavelet trees. The wavelet tree, invented by Grossi, Gupta, and Vitter [19], is an important data structure with a vast number of applications to stringology, computational geometry, and others (see [27] for an excellent survey). Despite this, the problem of wavelet tree construction has not received much attention in the literature. For a string w of length n , one can derive a construction algorithm with $\mathcal{O}(n \log \sigma)$ running time directly from the definition. Apart from this, two recent works [32, 10] present construction algorithms in the setting when limited extra space is allowed. Running time of the algorithms is higher than that of the naive algorithm. The first result of our paper is a novel deterministic algorithm for constructing a wavelet tree in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time.

Standard wavelet trees form a perfect binary tree with σ nodes, but different shapes have been introduced for several applications: among others this includes wavelet trees for Huffman encoding [17] and wavelet tries [20] that have the shape of a trie for a given set of strings. Our construction algorithm is capable of building arbitrary-shaped (binary) wavelet trees of $\mathcal{O}(\log \sigma)$ height with constant overhead.

Our algorithm for wavelet trees helps to derive some improved bounds for range rank/select queries. Namely, given an array $A[1..n]$ of integers, one could ask either to compute the number of integers in $A[i..j]$ that are smaller than a given $A[k]$ (*rank* query); or to find, for given i, j , and k , the k -th smallest integer in $A[i..j]$ (*select* query).

These problems have been widely studied; see e.g. Chan and Pătraşcu [8] and Brodal et al. [6]. By slightly tweaking our construction of wavelet trees, we can build in $\mathcal{O}(n \sqrt{\log n})$ deterministic time an $\mathcal{O}(n)$ size data structure for answering rank/select queries in $\mathcal{O}(\frac{\log n}{\log \log n})$ time. Our approach yields a $\sqrt{\log n}$ -factor improvement to the construction time upon Brodal et al. [6] and a $\log \log n$ -factor improvement to the query time upon Chan and Pătraşcu [8].

Wavelet suffix trees. Then we switch to stringological context and extend our approach to the so-called *internal* string problems (see [25]). This type of problems involves construction of a compact data structure for a given fixed string w capable of answering certain queries for substrings of w . This line of development was originally inspired by suffix trees, which can answer some basic internal string queries (e.g. equality testing and longest common extension computation) in constant time and linear space. Lately a number of studies emerged addressing compressibility [11, 23], range longest common prefixes (range LCP) [1, 28], periodicity [24, 13], minimal/maximal suffixes [4, 3], substring hashing [15, 18], and fragmented pattern matching [2, 18].

Our work focuses on rank/select problems for suffixes of a given substring. Given a fixed string w of length n , the goal is to preprocess it into a compact data structure for answering the following two types of queries efficiently:

- 1) *Substring suffix rank*: For substrings x and y of w (given by their endpoints) count the number of suffixes of x that are lexicographically smaller than y ;

- 2) *Substring suffix select*: For a substring x of w (given by its endpoints) and an integer k , find the k -th lexicographically smallest suffix of x .

Note that for $k = 1$ and $k = |x|$ substring suffix select queries reduce to computing the lexicographically minimal and the lexicographically maximal suffixes of a given substring. Study of this problem was started by Duval [14]. He showed that the maximal and the minimal suffixes of *all prefixes* of a string can be found in linear time and constant additional space. Later this problem was addressed in [4, 3]. In [3] it was shown that the minimal suffix of any substring can be computed in $\mathcal{O}(\tau)$ time by a linear space data structure with $\mathcal{O}(n \log n / \tau)$ construction time for any parameter τ , $1 \leq \tau \leq \log n$. As an application of this result it was shown how to construct the Lyndon decomposition of any substring in $\mathcal{O}(\tau s)$ time, where s stands for the number of distinct factors in the decomposition. For the maximal suffixes an optimal linear-space data structure with $\mathcal{O}(1)$ query and $\mathcal{O}(n)$ construction time was presented. We also note that [26] considered a problem with a similar name, namely substring rank and select. However, the goal there is to preprocess a string, so that given any pattern, we can count its occurrences in a specified prefix of the string, and select the k -th occurrence in the whole string. One can easily see that this problem is substantially different than the one we are interested in.

Here, we both generalize the problem to an arbitrary k (thus enabling to answer general *substring suffix select* queries) and also consider *substring suffix rank* queries. We devise a linear-space data structure with $\mathcal{O}(n\sqrt{\log n})$ expected construction time supporting both types of the queries in $\mathcal{O}(\log |x|)$ time.

Our approach to substring suffix rank/select problems is based on wavelet trees and attracts a number of additional combinatorial and algorithmic ideas. Combining wavelet trees with suffix trees we introduce a novel concept of *wavelet suffix trees*, which forms a foundation of our data structure. Like usual suffix trees, wavelet suffixes trees provide a compact encoding for all substrings of a given text; like wavelet trees they maintain logarithmic height. Our hope is that these properties will make wavelet suffix trees an attractive tool for a large variety of stringological problems.

We conclude with an application of wavelet suffix trees to substring compression, a class of problems introduced by Cormode and Muthukrishnan [11]. Queries of this type ask for a compressed representation of a given substring. The original paper, as well as a more recent work by Keller et al. [23], considered Lempel-Ziv compression schemes. Another family of methods, based on Burrows-Wheeler transform [7] was left open for further research. We show that wavelet suffix trees allow to compute a run-length-encoded Burrows-Wheeler transform of an arbitrary substring x of w (again, given by its endpoints) in $\mathcal{O}(s \log |x|)$ time, where s denotes the length of the resulting run-length encoding.

2 Wavelet Trees

Given a string s of length n over an alphabet $\Sigma = [0, \sigma - 1]$, where $\sigma \leq n$, we define the wavelet tree of s as follows. First, we create the root node r and construct its bitmask B_r of length n . To build the bitmask, we think of every $s[i]$ as of a binary number consisting of exactly $\log \sigma$ bits (to make the description easier to follow, we assume that σ is a power of 2), and put the most significant bit of $s[i]$ in $B_r[i]$. Then we partition s into s_0 and s_1 by scanning through s and appending $s[i]$ with the most significant bit removed to either s_1 or s_0 , depending on whether the removed bit of $s[i]$ was set or not, respectively. Finally, we recursively define the wavelet trees for s_0 and s_1 , which are strings over the alphabet $[0, \sigma/2 - 1]$, and attach these trees to the root. We stop when the

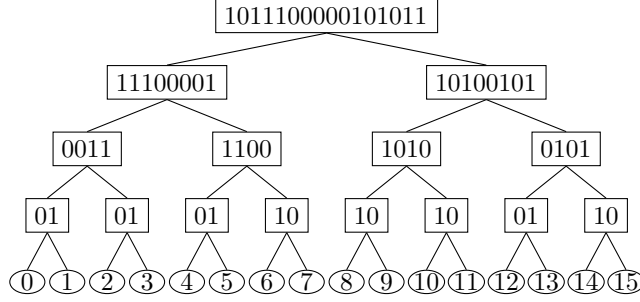


Figure 1: Wavelet tree for a string 1100₂ 0111₂ 1011₂ 1111₂ 1001₂ 0110₂ 0100₂ 0000₂ 0001₂ 0010₂ 1010₂ 0011₂ 1101₂ 0101₂ 1000₂ 1110₂, the leaves are labelled with their corresponding characters.

alphabet is unary. The final result is a perfect binary tree on σ leaves with a bitmask attached at every non-leaf node. Assuming that the edges are labelled by **0** or **1** depending on whether they go to the left or to the right respectively, we can define a *label* of a node to be the concatenation of the labels of the edges on the path from the root to this node. Then leaf labels are the binary representations of characters in $[0, \sigma - 1]$; see Figure 1 for an example.

In virtually all applications, each bitmask B_r is augmented with a rank/select structure. For a bitmask $B[1..N]$ this structure implements operations $\text{rank}_b(i)$, which counts the occurrences of a bit $b \in \{0, 1\}$ in $B[1..i]$, and $\text{select}_b(i)$, which selects the i -th occurrence of b in the whole $B[1..n]$, for any $b \in \{0, 1\}$, both in $\mathcal{O}(1)$ time.

The bitmasks and their corresponding rank/select structures are stored one after another, each starting at a new machine word. The total space occupied by the bitmasks alone is $\mathcal{O}(n \log \sigma)$ bits, because there are $\log \sigma$ levels, and the lengths of the bitmasks for all nodes at one level sum up to n . A rank/select structure built for a bit string $B[1..N]$ takes $o(N)$ bits [9, 21], so the space taken by all of them is $o(n \log \sigma)$ bits. Additionally, we might lose one machine word per node because of the word alignment, which sums up to $\mathcal{O}(\sigma)$. For efficient navigation, we number the nodes in a heap-like fashion and, using $\mathcal{O}(\sigma)$ space, store for every node the offset where its bitmasks and the corresponding rank/select structure begin. Thus, the total size of a wavelet tree is $\mathcal{O}(\sigma + n \log \sigma / \log n)$, which for $\sigma \leq n$ is $\mathcal{O}(n \log \sigma / \log n)$.

Directly from the recursive definition, we get a construction algorithm taking $\mathcal{O}(n \log \sigma)$ time, but we can hope to achieve a better time complexity as the size of the output is just $\mathcal{O}(n \log \sigma / \log n)$ words. In Section 2.1 we show that one can construct all bitmasks augmented with the rank/select structures in total time $\mathcal{O}(n \log \sigma / \sqrt{\log n})$.

Arbitrarily-shaped wavelet trees. A generalized view on a wavelet tree is that apart from a string s we are given an arbitrary full binary tree T (i.e., a rooted tree whose nodes have 0 or 2 children) on σ leaves, together with a bijective mapping between the leaves and the characters in Σ . Then, while defining the bitmasks B_v , we do not remove the most significant bit of each character, and instead of partitioning the values based on this bit, we make a decision based on whether the leaf corresponding to the character lies in the left or in the right subtree of v . Our construction algorithm generalizes to such arbitrarily-shaped wavelet trees without increasing the time complexity provided that the height of T is $\mathcal{O}(\log \sigma)$.

Wavelet trees of larger degree. Binary wavelet trees can be generalized to higher degree trees in a natural way as follows. We think of every $s[i]$ as of a number in base d . The degree- d wavelet tree of s is a tree on σ leaves, where every inner node is of degree d , except for the last level, where the nodes may have smaller degrees. First, we create its root node r and construct its string D_r of length n setting as $D_r[i]$ the most significant digit of $s[i]$. We partition s into d strings s_0, s_1, \dots, s_{d-1} by scanning through s and appending $s[i]$ with the most significant digit removed to s_a , where a is the removed digit of $s[i]$. We recursively repeat the construction for every s_a and attach the resulting tree as the a -th child of the root. All strings D_u take $\mathcal{O}(n \log \sigma)$ bits in total, and every D_u is augmented with a generalized rank/select structure.

We consider $d = \log^\varepsilon n$, for a small constant $\varepsilon > 0$. Remember that we assume σ to be a power of 2, and because of similar technical reasons we assume d to be a power of two as well. Under such assumptions, our construction algorithm for binary wavelet trees can be used to construct a higher degree wavelet tree. More precisely, in Section 2.3 we show how to construct such a higher degree tree in $\mathcal{O}(n \log \log n)$, assuming that we are already given the binary wavelet tree, making the total construction time $\mathcal{O}(n \sqrt{\log n})$ if $\sigma \leq n$, and allowing us to derive improved bounds on range selection, as explained in Section 2.3.

2.1 Wavelet Tree Construction

Given a string s of length n over an alphabet Σ , we want to construct its wavelet tree, which requires constructing the bitmasks and their rank/select structures, in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time.

Input representation. The input string s can be considered as a list of $\log \sigma$ -bit integers. We assume that it is represented in a packed form, as described below.

A single machine word of length $\log n$ can accommodate $\frac{\log n}{b}$ b -bit integers. Therefore a list of N such integers can be stored in $\frac{Nb}{\log n}$ machine words. (For s , $b = \log \sigma$, but later we will also consider other values of b .) We call such a representation a *packed list*.

We assume that packed lists are implemented as resizable bitmasks (with each block of b bits representing a single entry) equipped with the size of the packed list. This way a packed list of length N can be appended to another packed list in $\mathcal{O}(1 + \frac{Nb}{\log n})$ time, since our model supports bit-shifts in $\mathcal{O}(1)$ time. Similarly, splitting into lists of length at most k takes $\mathcal{O}(\frac{Nb}{\log n} + \frac{N}{k})$ time.

Overview. Let τ be a parameter to be chosen later to minimize the total running time. We call a node u *big* if its depth is a multiple of τ , and *small* otherwise. The construction conceptually proceeds in two phases.

First, for every big node u we build a list S_u . Remember that the label ℓ_u of a node u at distance α from the root consists of α bits, and the characters corresponding to the leaves below u are exactly the characters whose binary representation starts with ℓ_u . S_u is defined to be a subsequence of s consisting of these characters.

Secondly, we construct the bitmasks B_v for every node v using S_u of the nearest big ancestor u of v (possibly v itself).

First phase. Assume that for a big node u at distance $\alpha\tau$ from the root we are given the list S_u . We want to construct the lists S_v for all big nodes v whose deepest (proper) big ancestor is u . There are exactly 2^τ such nodes v , call them $v_0, \dots, v_{2^\tau-1}$. To construct their lists S_{v_i} , we scan

through S_u and append $S_u[j]$ to the list of v_t , where t is a bit string of length τ occurring in the binary representation of $S_u[j]$ at position $\alpha\tau + 1$. We can extract t in constant time, so the total complexity is linear in the total number of elements on all lists, i.e., $\mathcal{O}(n \log \sigma / \tau)$ if $\tau \leq \log \sigma$. Otherwise the root is the only big node, and the first phase is void.

Second phase. Assume that we are given the list S_u for a big node u at distance $\alpha\tau$ from the root. We would like to construct the bitmask B_v for every node v such that u is the nearest big ancestor of v . First, we observe that to construct all these bitmasks we only need to know τ bits of every $S_u[j]$ starting from the $(\alpha\tau + 1)$ -th one. Therefore, we will operate on *short lists* L_v consisting of τ -bit integers instead of the lists S_v of whole $\log \sigma$ -bit integers. Each short list is stored as a packed list.

We start with extracting the appropriate part of each $S_u[j]$ and appending it to L_u . The running time of this step is proportional to the length of L_u . (This step is void if $\tau > \log \sigma$; then $L_v = S_v$.) Next, we process all nodes v such that u is the deepest big ancestor of v . For every such v we want to construct the short list L_v . Assuming that we already have L_v for a node v at distance $\alpha\tau + \beta$ from the root, where $\beta \in [0, \tau)$, and we want to construct the short lists of its children and the bitmask B_v . This can be (inefficiently) done by scanning L_v and appending the next element to the short list of the right or the left child of v , depending on whether its $(\beta + 1)$ -th bit is set or not, respectively. The bitmask B_v simply stores all these $(\beta + 1)$ -th most significant bits. In order to compute these three lists efficiently we apply the following claim.

Claim 2.1. *Assuming $\tilde{\mathcal{O}}(\sqrt{n})$ space and preprocessing shared by all instances of the structure, the following operation can be performed in $\mathcal{O}(\frac{Nb}{\log n})$ time: given a packed list L of N b -bit integers, where $b = o(\log n)$, and a position $t \in [0, b - 1]$, compute packed lists L_0 and L_1 consisting of the elements of L whose t -th most significant bit is 0 or 1, respectively, and a bitmask B being a concatenation of the t -th most significant bits of the elements of L .*

Proof. As a preprocessing, we precompute all the answers for lists L of length at most $\frac{1}{2} \frac{\log n}{b}$. This takes $\tilde{\mathcal{O}}(\sqrt{n})$ time. For a query we split L into lists of length $\frac{1}{2} \frac{\log n}{b}$, apply the preprocessed mapping and merge the results, separately for L_0, L_1 and B . This takes $\mathcal{O}(\frac{Nb}{\log n})$ time in total. \square

Consequently, we spend $\mathcal{O}(|L_v|\tau / \log n)$ per node v . The total number of the elements of all short lists is $\mathcal{O}(n \log \sigma)$, but we also need to take into the account the fact that the lengths of some short lists might be not divisible by $\log n / \tau$, which adds $\mathcal{O}(1)$ time per a node of the wavelet tree, making the total complexity $\mathcal{O}(\sigma + n \log \sigma \tau / \log n) = \mathcal{O}(n \log \sigma \tau / \log n)$.

Intermixing the phases. In order to make sure that space usage of the construction algorithm does not exceed the size of the the final structure, the phases are intermixed in the following way. Assuming that we are given the lists S_u of all big nodes u at distance $\alpha\tau$ from the root, we construct the lists of all big nodes at distance $(\alpha + 1)\tau$ from the root, if any. Then we construct the bitmasks B_u such that the nearest big ancestor of u is at distance $\alpha\tau$ from the root and increase α . To construct the bitmasks, we compute the short lists for all nodes at distances $\alpha\tau, \dots, (\alpha + 1)\tau - 1$ from the root and keep the short lists only for the nodes at the current distance. Then the peak space of the construction process is just $\mathcal{O}(n \log \sigma / \log n)$ words.

The total construction time is $\mathcal{O}(n \log \sigma / \tau)$ for the first phase and $\mathcal{O}(n \log \sigma \tau / \log n)$ for the second phase. The bitmasks, lists and short lists constructed by the algorithm are illustrated in Figure 2. Choosing $\tau = \sqrt{\log n}$ as to minimize the total time, we get the following theorem.

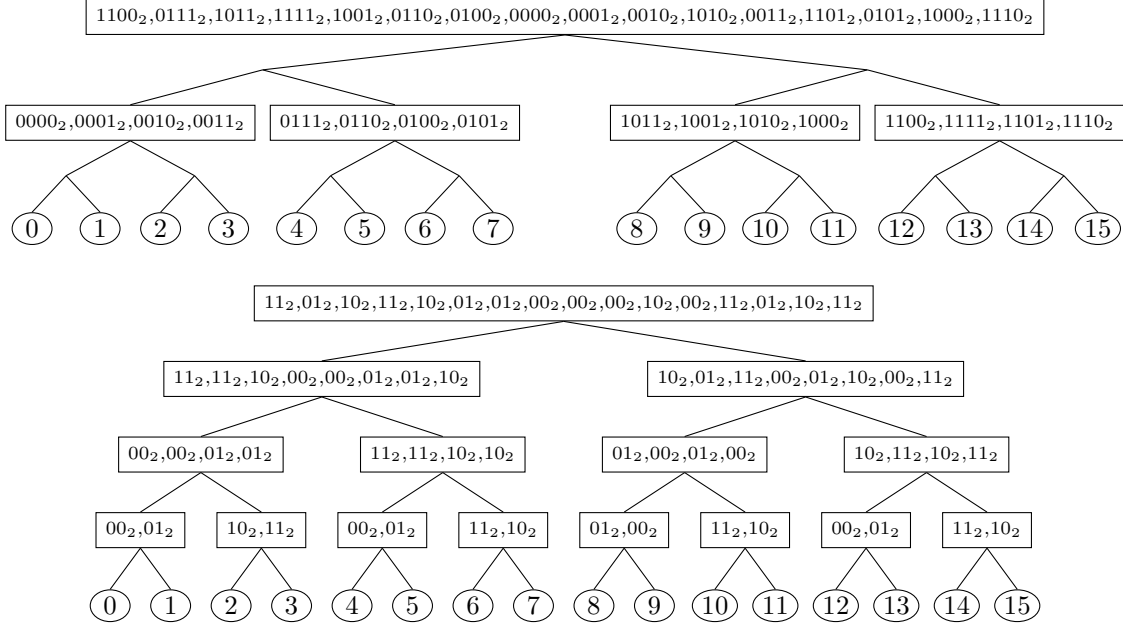


Figure 2: Elements of the wavelet tree construction algorithm for the string from Figure 1. The first figure shows the lists S_u of all big nodes u when $\tau = 2$, and the third shows the short lists L_u of all nodes u , as defined in the proof of Theorem 2.2.

Theorem 2.2. *Given a string s of length n over an alphabet Σ , we can construct all bitmasks B_u of its wavelet tree in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time.*

Additionally, we want to build a rank/select structure for every B_u . While it is well-known that given a bit string of length N one can construct an additional structure of size $o(N)$ allowing executing both rank and select in $\mathcal{O}(1)$ time [9, 21], we must verify that the construction time is not too high. The following lemma is proved in Appendix A.

Lemma 2.3. *Given a bit string $B[1..N]$ packed in $\frac{N}{\log n}$ machine words, we can extend it in $\mathcal{O}(\frac{N}{\log n})$ time with a rank/select structure occupying additional $o(\frac{N}{\log n})$ space, assuming an $\tilde{\mathcal{O}}(\sqrt{n})$ time and space preprocessing shared by all instances of the structure.*

2.2 Arbitrarily-Shaped Wavelet Trees

To generalize the algorithm to arbitrarily-shaped wavelet trees of degree $\mathcal{O}(\log \sigma)$, instead of operating on the characters we work with the labels of the root-to-leaf paths, appended with **0**s so that they all have the same length. The heap-like numbering of the nodes is not enough in such setting, as we cannot guarantee that all leaves have the same depth, so the first step is to show how to efficiently return a node given the length of its label and the label itself stored in a single machine word. If $\log \sigma = o(\log n)$ we can afford storing nodes in a simple array, otherwise we use the deterministic dictionary of Ružić [31], which can be constructed in $\mathcal{O}(\sigma(\log \log \sigma)^2) = \mathcal{O}(n(\log \log n)^2)$ time. In either case, we can return in $\mathcal{O}(1)$ time the corresponding pointer, which might be null if

the node does not exist. Then the construction algorithm works as described previously: first we construct the list S_u of every big node u , and then we construct every B_v using the S_u of the nearest big ancestor of v . The only difference is that when splitting the list S_u into the lists S_v of all big nodes v such that u is the first proper big ancestor of v , it might happen that the retrieved pointer to v is null. In such case we simply continue without appending anything to the non-existing S_v . The running time stays the same.

Theorem 2.4. *Let s be a string of length n over Σ and T be a full binary tree of height $\mathcal{O}(\log \sigma)$ with σ leaves, each assigned a distinct character in Σ . Then the T -shaped wavelet tree of s can be constructed in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time.*

2.3 Wavelet Trees of Larger Degree

We move to constructing a wavelet tree of degree $d = \log^\varepsilon n$, where d is a power of two. Such higher degree tree can be also defined through the binary wavelet tree for s as follows. We remove all inner nodes whose depth is not a multiple of $\log d$. For each surviving node we set its nearest preserved ancestor as a parent. Then each inner node has d children (the lowest-level nodes may have fewer children), and we order them consistently with the left-to-right order in the original tree.

Recall that for each node u of the binary wavelet tree we define the string S_u as a subsequence of s consisting of its characters whose binary representation starts with the label ℓ_u of u . Then we create the bitmask storing, for each character of S_u , the bit following its label ℓ_u . Instead of B_u a node u of the wavelet tree of degree d now stores a string D_u , which contains the next $\log d$ bits following ℓ_u .

The following lemma allows to use binary wavelet tree construction as a black box, and consequently gives an $\mathcal{O}(n\sqrt{\log n})$ -time construction algorithm for wavelet trees of degree d .

Lemma 2.5. *Given all bitmasks B_u , we can construct all strings D_u in $\mathcal{O}(n \log \log n)$ time.*

Proof. Consider a node u at depth k of the wavelet tree of degree d . Its children correspond to all descendants of u in the original wavelet tree at depth $(k+1)d$. For each descendant v of u at depth $(k+1)\log d - \delta$, where $\delta \in [0, \log d]$, we construct a temporary string D'_v over the alphabet $[0, 2^\delta - 1]$. Every character of this temporary string corresponds to a leaf in the subtree of v . The characters are arranged in order of the identifiers of the corresponding leaves and describe prefixes of length δ of paths from u to the leaves. Clearly $D_u = D'_u$. Furthermore, if the children of v are v_1 and v_2 , then D'_v can be easily defined by looking at D'_{v_1} , D'_{v_2} , and B_v as follows. We prepend **0** to all characters in D'_{v_1} , and **1** to all characters in D'_{v_2} . Then we construct D'_v by appropriately interleaving D'_{v_1} and D'_{v_2} according to the order defined by B_v . We consider the bits of B_v one-by-one. If the i -th bit is **0**, we append the next character of D'_{v_1} to the current D'_v , and otherwise we append the next character of D'_{v_2} . Now we only have to show to implement this process efficiently.

We pack every $\frac{1}{4} \frac{\log n}{\log d}$ consecutive characters of D'_v into a single machine word. To simplify the implementation, we reserve $\log d$ bits for every character irrespectively of the value of δ . This makes prepending **0**s or **1**s to all characters in any D'_v trivial, because the result can be preprocessed in $\tilde{\mathcal{O}}(d^{\frac{1}{4} \frac{\log n}{\log d}}) = o(n)$ time and space. Interleaving D'_{v_1} and D'_{v_2} is more complicated. Instead of accessing D'_{v_1} and D'_{v_2} directly, we keep two buffers, each containing at most next $\frac{1}{4} \frac{\log n}{\log d}$ characters from the corresponding string. Similarly, instead of accessing B_v directly, we keep a buffer of at most $\frac{1}{4} \log n$ next bits there. Using the buffers, we can keep processing bits from B_v as long as there are enough characters in the input buffers. The input buffers for D'_{v_1} and D'_{v_2} become empty

after generating $\frac{1}{4} \frac{\log n}{\log d}$ characters, and the input buffer for B_v becomes empty after generating $\frac{1}{4} \log n$ characters. Hence the total number of times we need to reload one of the input buffers is $\mathcal{O}(|D'_v| / \frac{\log n}{\log d})$.

We preprocess all possible scenarios between two reloads by simulating, for every possible initial content of the input buffers, processing the bits until one of the buffers becomes empty. We store the generated data (which is at most $\frac{1}{2} \log n$ bits) and the final content of the input buffers. The whole preprocessing takes $\tilde{\mathcal{O}}(2^{\frac{3}{4} \log n}) = o(n)$ time and space, and then the number of operations required to generate packed D'_v is proportional to the number of times we need to reload the buffers, so by summing over all v the total complexity is $\mathcal{O}(n \log \log n)$. \square

Then we extend every D_u with a generalized rank/select data structure. Such a structure for a string $D[1..N]$ implements operations $\text{rank}_c(i)$, which counts positions $k \in [1, i]$ such that $D[k] \leq c$, and $\text{select}_c(i)$, which selects the i -th occurrence of c in the whole $D[1..n]$, for any $c \in \Sigma$, both in $\mathcal{O}(1)$ time. Again, it is well-known that such a structure can be implemented using just $o(n \log \sigma)$ additional bits if $\sigma = \mathcal{O}(\text{polylog}(n))$ [16], but its construction time is not explicitly stated in the literature. Therefore, we prove the following lemma in Appendix A.

Lemma 2.6. *Let $d \leq \log^\varepsilon n$ for $\varepsilon < \frac{1}{3}$. Given a string $D[1..N]$ over the alphabet $[0, d-1]$ packed in $\frac{N \log d}{\log n}$ machine words, we can extend it in $\mathcal{O}(\frac{N \log d}{\log n})$ time with a generalized rank/select data structure occupying additional $o(\frac{N}{\log n})$ space, assuming $\tilde{\mathcal{O}}(\sqrt{n})$ time and space preprocessing shared by all instances of the structure.*

2.4 Range Selection

A classic application of wavelet trees is that, given an array $A[1..n]$ of integers, we can construct a structure of size $\mathcal{O}(n)$, which allows answering any range rank/select query in $\mathcal{O}(\log n)$ time. A range select query is to return the k -th smallest element in $A[i..j]$, while a range rank query is to count how many of $A[i..j]$ are smaller than given $x = A[k]$. Given the wavelet tree for A , any range rank/select query can be answered by traversing a root-to-leaf path of the tree using the rank/select data structures for bitmasks B_u at subsequent nodes.

With $\mathcal{O}(n\sqrt{\log n})$ construction algorithm this matches the bounds of Chan and Pătraşcu [8] for range select queries, but is slower by a factor of $\log \log n$ than their solution for range rank queries. We will show that one can in fact answer any range rank/select query in $\mathcal{O}(\frac{\log n}{\log \log n})$ time with an $\mathcal{O}(n)$ size structure, which can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time. For range rank queries this is not a new result, but we feel that our proof gives more insight into the structure of the problem. For range select queries, Brodal et al. [6] showed that one can answer a query in $\mathcal{O}(\frac{\log n}{\log \log n})$ time with an $\mathcal{O}(n)$ size structure, but with $\mathcal{O}(n \log n)$ construction time. Chan and Pătraşcu asked if methods of [7] can be combined with the efficient construction. We answer this question affirmatively.

A range rank query is easily implemented in $\mathcal{O}(\frac{\log n}{\log \log n})$ time using wavelet tree of degree $\log^\varepsilon n$ described in the previous section. To compute the rank of $x = A[k]$ in $A[i..j]$, we traverse the path from the root to the leaf corresponding to $A[k]$. At every node we use the generalized rank structure to update the answer and the current interval $[i..j]$ before we descend.

Implementing the range select queries in $\mathcal{O}(\frac{\log n}{\log \log n})$ time is more complicated. Similar to the rank queries, we start the traverse at the root and descend along a root-to-leaf path. At each node we select its child we will descend to, and update the current interval $[i..j]$ using the generalized rank data structure. To make this query algorithm fast, we preprocess each string D_u and store

extracted information in matrix form. As shown by Brodal et al. [6], constant time access to such information is enough to implement range select queries in $\mathcal{O}(\log n / \log \log n)$ time.

The matrices for a string D_u are defined as follows. We partition D_u into superblocks of length $d \log^2 n$.¹ For each superblock we store the cumulative generalized rank of every character, i.e., for every character c we store the number positions where characters $c' \leq c$ occur in the prefix of the string up to the beginning of the superblock. We think of this as a $d \times \log n$ matrix M . The matrix is stored in two different ways. In the first copy, every row is stored as a single word. In the second copy, we divide the matrix into sections of $\log n / d$ columns, and store every section in a single word. We make sure there is an overlap of four columns between the sections, meaning that the first section contains columns $1, \dots, \log n / d$, the second section contains columns $\log n / d - 3, \dots, 2 \log n / d - 4$, and so on.

Each superblock is then partitioned into blocks of length $\log n / \log d$ each. For every block, we store the cumulative generalized rank within the superblock of every character, i.e., for every character c we store the number of positions where characters $c' \leq c$ occur in the prefix of the superblock up to the end beginning of the block. We represent this information in a *small matrix* M' , which can be packed in a single word, as it requires only $\mathcal{O}(d \log(d \log^2 n))$ bits.

Lemma 2.7. *Given a string $D[1..N]$ over the alphabet $[0, d - 1]$ packed in $\frac{N \log d}{\log n}$ machine words, we can extend it in $\mathcal{O}(\frac{N \log d}{\log n})$ time and space with the following information:*

- 1) *The first copy of the matrix M for each superblock (multiple of $d \log^2 n$);*
- 2) *The second copy of the matrix M for each superblock (multiple of $d \log^2 n$);*
- 3) *The small matrix M' for each block (multiple of $\log n / d$);*

occupying additional $\mathcal{O}(\frac{N \log d}{\log n})$ space, assuming an $\tilde{\mathcal{O}}(\sqrt{n})$ time and space preprocessing shared by all instances of the structure and $d = \log^\varepsilon n$.

Proof. To compute the small matrices M' , i.e., the cumulative generalized ranks for blocks, and the first copies of the matrices M , i.e., the cumulative generalized ranks for superblocks, we notice that the standard solution for generalized rank queries in $\mathcal{O}(1)$ time is to split the string into superblocks and blocks. Then, for every superblock we store the cumulative generalized rank of every character, and for every block we store the cumulative generalized rank within the superblock for every character. As explained in the proof of Lemma 2.6 presented in the appendix, such data can be computed in $\mathcal{O}(\frac{N \log d}{\log n})$ time if the size of the blocks and the superblocks are chosen to be $\frac{1}{3} \frac{\log n}{\log d}$ and $d \log^2 n$, respectively. Therefore, we obtain in the same complexity every first copy of the matrix M , and a small matrix every $\frac{1}{3} \frac{\log n}{\log d}$ characters. We can simply extract and save every third such small matrix, also in the same complexity.

The second copies of the matrices are constructed from the first copies in $\mathcal{O}(d^2)$ time each; we simply partition each row into (overlapping) sections and append each part to the appropriate machine words. This takes $\mathcal{O}(\frac{Nd^2}{d^2 \log n}) = \mathcal{O}(\frac{N}{\log n})$ in total. \square

As follows from the lemma, all strings D_u at a single level of the tree can be extended in $\mathcal{O}(n \log \log n / \log n)$ time, which, together with constructing the tree itself, sums up to $\mathcal{O}(n \sqrt{\log n})$ time in total.

¹In the original paper, superblocks are of length $d \log n$, but this does not change the query algorithm.

3 Wavelet Suffix Trees

In this section we generalize wavelet trees to obtain wavelet suffix trees. With logarithmic height and shape resembling the shape of the suffix tree wavelet suffix trees, augmented with additional stringological data structures, become a very powerful tool. In particular, they allow to answer the following queries efficiently: (1) find the k -th lexicographically minimal suffix of a substring of the given string (*substring suffix selection*), (2) find the rank of one substring among the suffixes of another substring (*substring suffix rank*), and (3) compute the run-length encoding of the Burrows-Wheeler transform of a substring.

Organization of Section 3 In Section 3.1 we introduce several, mostly standard, stringological notions and recall some already known algorithmic results. Section 3.2 provides a high-level description of the wavelet suffix trees. It forms an interface between the query algorithms (Section 3.5) and the more technical content: full description of the data structure (Section 3.3) and its construction algorithm (Section 3.4). Consequently, Sections 3.3 and 3.5 can be read separately. The latter additionally contains cell-probe lower bounds for some of the queries (suffix rank & selection), as well as a description of a generic transformation of the data structure, which allows to replace a dependence on n with a dependence on $|x|$ in the running times of the query algorithms.

3.1 Preliminaries

Let w be a string of length $|w| = n$ over the alphabet $\Sigma = [0, \sigma - 1]$. For $1 \leq i \leq j \leq n$, $w[i..j]$ denotes the *substring* of w from position i to position j (inclusive). For $i = 1$ or $j = |w|$, we use shorthands $w[..j]$ and $w[i..]$. If $x = w[i..j]$, we say that x *occurs* in w at position i . Each substring $w[..j]$ is called a *prefix* of w , and each substring $w[i..]$ is called a *suffix* of w . A substring which occurs both as a prefix and as a suffix of w is called a *border* of w . The length longest common prefix of two strings x, y is denoted by $\text{lcp}(x, y)$.

We extend Σ with a sentinel symbol $\$$, which we assume to be smaller than any other character. The order on Σ can be generalized in a standard way to the *lexicographic* order of the strings over Σ : a string x is lexicographically smaller than y (denoted $x \prec y$) if either x is a proper prefix, or there exists a position i , $0 \leq i < \min\{|x|, |y|\}$, such that $x[1..i] = y[1..i]$ and $x[i+1] \prec y[i+1]$. The following lemma provides one of the standard tools in stringology.

Lemma 3.1 (LCP Queries [12]). *A string w of length n can be preprocessed in $\mathcal{O}(n)$ time so that the following queries can be answered in $\mathcal{O}(1)$ time: Given substrings x and y of w , compute $\text{lcp}(x, y)$ and decide whether $x \prec y$, $x = y$, or $x \succ y$.*

We say that a sequence $p_0 < p_1 < \dots < p_k$ of positions in a string w is a *periodic progression* if $w[p_0..p_1 - 1] = \dots = w[p_{k-1}..p_k - 1]$. Periodic progressions p, p' are called *non-overlapping* if the maximum term in p is smaller than the minimum term in p' or vice versa, the maximum term in p' is smaller than the minimum term in p . Note that any periodic progression is an arithmetic progression and consequently it can be represented by three integers: p_0 , $p_1 - p_0$, and k . Periodic progressions appear in our work because of the following result:

Theorem 3.2 ([25]). *Using a data structure of size $\mathcal{O}(n)$ with $\mathcal{O}(n)$ -time randomized (Las Vegas) construction, the following queries can be answered in constant time: Given two substrings x and y such that $|x| = \mathcal{O}(|y|)$, report the positions of all occurrences of y in x , represented as at most $\frac{|x|+1}{|y|+1}$ non-overlapping periodic progressions.*

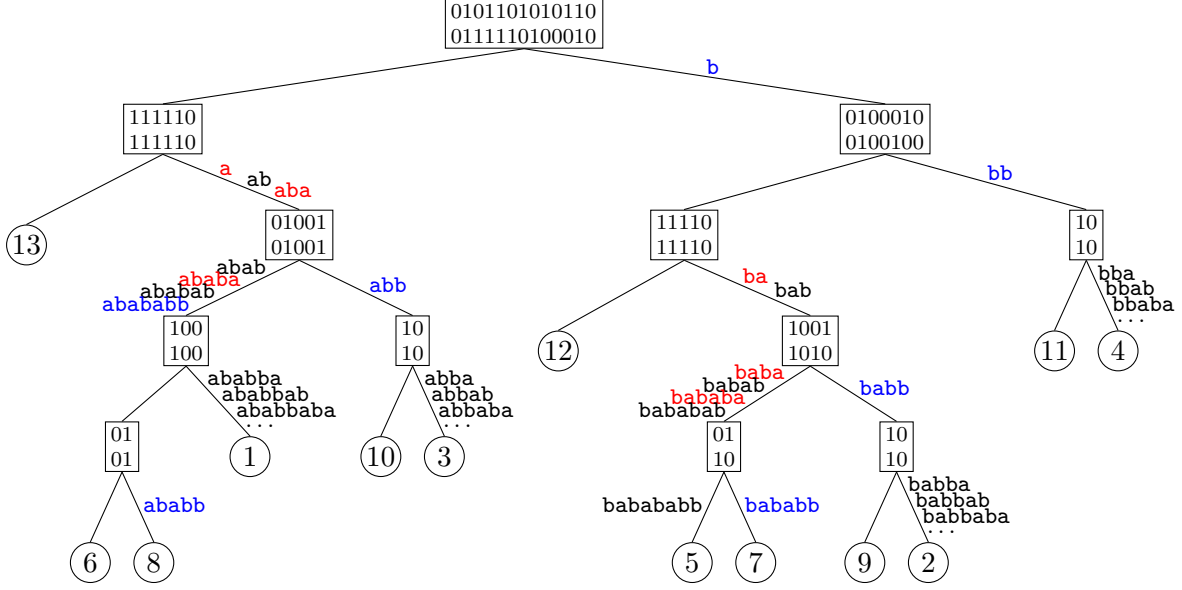


Figure 3: A wavelet suffix tree of $w = \text{ababbabababb}$. Leaves corresponding to $w[i..]\$$ are labelled with i . Elements of $L(e)$ are listed next to e , with \dots denoting further substrings up to the suffix of w . Suffixes of $x = \text{bababa}$ are marked red, of $x = \text{abababb}$: blue. Note that the prefixes of $w[i..]$ do not need to lie above the leaf i (see $w[1,5] = \text{ababb}$), and the substrings above the leaf i do not need to be prefixes of $w[i..]$ (see $w[10..]$ and aba).

3.2 Overview of wavelet suffix trees

For a string w of length n , a *wavelet suffix tree* of w is a full binary tree of logarithmic height. Each of its n leaves corresponds to a non-empty suffix of $w\$$. The lexicographic order of suffixes is preserved as the left-to-right order of leaves.

Each node u of the wavelet suffix tree stores two bitmasks. Bits of the first bitmask correspond to suffixes below u sorted by their starting positions, and bits of the second bitmask correspond to these suffixes sorted by pairs (preceding character, starting position). The i -th bit of either bitmask is set to 0 if the i -th suffix belongs to the left subtree of u and to 1 otherwise. Like in the standard wavelet trees, on top of the bitmasks we maintain a rank/select data structure. See Figure 3 for a sample wavelet suffix tree with both bitmasks listed down in nodes.

Each edge e of the wavelet suffix tree is associated with a sorted list $L(e)$ containing substrings of w . The wavelet suffix tree enjoys an important *lexicographic property*. Imagine we traverse the tree depth-first, and when going *down* an edge e we write out the contents of $L(e)$, whereas when visiting a leaf we output the corresponding suffix of $w\$$. Then, we obtain the lexicographically sorted list of all substrings of $w\$$ (without repetitions).² This, in particular, implies that the substrings in $L(e)$ are consecutive prefixes of the longest substring in $L(e)$, and that for each substring y of w there is exactly one edge e such that the $y \in L(e)$.

In the query algorithms, we actually work with $L_x(e)$, containing the suffixes of x among the elements of $L(e)$. For each edge e , starting positions of these suffixes form $\mathcal{O}(1)$ non-overlapping

²A similar property holds for suffix trees if we define $L(e)$ so that it contains the labels of all implicit nodes on e and the label of the lower explicit endpoint of e .

periodic progressions, and consequently the list $L_x(e)$ admits a constant-space representation. Nevertheless, we do not store the lists explicitly, but instead generate some of them on the fly. This is one of the auxiliary operations, each of which is supported by the wavelet suffix tree in constant time.

- (1) For a substring x and an edge e , output the list $L_x(e)$ represented as $\mathcal{O}(1)$ non-overlapping periodic progressions;
- (2) Count the number of suffixes of $x = w[i..j]$ in the left/right subtree of a node (given along with the segment of its first bitmask corresponding to suffixes that start inside $[i, j]$);
- (3) Count the number of suffixes $x = w[i..j]$ that are preceded by a character c and lie in the left/right subtree of a node (given along with the segment of its second bitmask corresponding to suffixes that start inside $[i, j]$ and are preceded by c);
- (4) For a substring x and an edge e , compute the run-length encoding of the sequence of characters preceding suffixes in $L_x(e)$.

3.3 Full description of wavelet suffix trees

We start the description with Section 3.3.1, where we introduce *string intervals*, a notion central to the definition of wavelet suffix tree. We also present there corollaries of Lemma 3.1 which let us efficiently deal with string intervals. Then, in Section 3.3.2, we give a precise definition of wavelet suffix trees and prove its several combinatorial consequences. We conclude with Section 3.3.3, where we provide the implementations of auxiliary operations defined in Section 3.2.

3.3.1 String intervals

To define wavelet suffix trees, we often need to compare substrings of w trimmed to a certain number of characters. If instead of x and y we compare their counterparts trimmed to ℓ characters, i.e., $x[1..\min\{\ell, |x|\}]$ and $y[1..\min\{\ell, |y|\}]$, we use ℓ in the subscript of the operator, e.g., $x =_\ell y$ or $x \preceq_\ell y$.

For a pair of strings s, t and a positive integer ℓ we define a *string interval* $[s, t]_\ell = \{z \in \bar{\Sigma}^* : s \preceq_\ell z \preceq_\ell t\}$ and $(s, t)_\ell = \{z \in \bar{\Sigma}^* : s \prec_\ell z \prec_\ell t\}$. Intervals $[s, t]_\ell$ and $(s, t)_\ell$ are defined analogously. The strings s, t are called the *endpoints* of these intervals.

In the remainder of this section, we show that the data structure of Lemma 3.1 can answer queries related to string intervals and periodic progressions, which arise in Section 3.3.3. We start with a simple auxiliary result; here y^∞ denotes a (one-sided) infinite string obtained by concatenating an infinite number of copies of y .

Lemma 3.3. *The data structure of Lemma 3.1 supports the following queries in $\mathcal{O}(1)$ time:*

- (1) *Given substrings x, y of w and an integer ℓ , determine if $x \prec_\ell y$, $x =_\ell y$, or $x \succ_\ell y$.*
- (2) *Given substrings x, y of w , compute $\text{lcp}(x, y^\infty)$ and determine whether $x \prec y^\infty$ or $x \succ y^\infty$.*

Proof. (1) By Lemma 3.1, we may assume to know $\text{lcp}(x, y)$. If $\text{lcp}(x, y) \geq \ell$, then $x =_\ell y$. Otherwise, trimming x and y to ℓ characters does not influence the order between these two substrings.

(2) If $\text{lcp}(x, y) < |y|$, i.e., y is not a prefix of x , then $\text{lcp}(x, y^\infty) = \text{lcp}(x, y)$ and the order between x and y^∞ is the same as between x and y . Otherwise, define x' so that $x = yx'$. Then $\text{lcp}(x, y^\infty) =$

$|y| + \text{lcp}(x', x)$ and the order between x and y^∞ is the same as between x' and x . Consequently, the query can be answered in constant time in both cases. \square

Lemma 3.4. *The data structure of Lemma 3.1 supports the following queries in $\mathcal{O}(1)$ time: Given a periodic progression $p_0 < \dots < p_k$ in w , a position $j \geq p_k$, and a string interval I whose endpoints are substrings of w , report, as a single periodic progression, all positions p_i such that $w[p_i..j] \in I$.*

Proof. If $k = 0$, it suffices to apply Lemma 3.3(1). Thus, we assume $k \geq 1$ in the remainder of the proof.

Let s and t be the endpoints of I , $\rho = w[p_0..p_1 - 1]$, and $x_i = w[p_i..j]$. Using Lemma 3.3(2) we can compute $r_0 = \text{lcp}(x_0, \rho^\infty)$ and $r' = \text{lcp}(s, \rho^\infty)$. Note that $r_i := \text{lcp}(x_i, \rho^\infty) = r - i|\rho|$, in particular $r_0 \geq k|\rho|$. If $r' \geq \ell$, we distinguish two cases:

- 1) $r_i \geq \ell$. Then $\text{lcp}(x_i, s) \geq \ell$; thus $x_i =_\ell s$.
- 2) $r_i < \ell$. Then $\text{lcp}(x_i, s) = r_i$; thus $x_i \prec_\ell s$ if $x_0 \prec \rho^\infty$, and $x_i \succ_\ell s$ otherwise.

On the other hand, if $r' < \ell$, we distinguish three cases:

- 1) $r_i > r'$. Then $\text{lcp}(x_i, s) = r'$; thus $x_i \prec_\ell s$ if $\rho^\infty \prec s$, and $x_i \succ_\ell s$ otherwise.
- 2) $r_i = r'$. Then we use Lemma 3.3(2) to determine the order between x_i and s trimmed to ℓ characters. This, however, may happen only for a single value i .
- 3) $r_i < r'$. Then $\text{lcp}(x_i, s) = r_i$; thus $x_i \prec_\ell s$ if $x_0 \prec \rho^\infty$, and $x_i \succ_\ell s$ otherwise.

Consequently, in constant time we can partition indices i into at most three ranges, and for each range determine whether $x_i \prec_\ell s$, $x_i =_\ell s$, or $x_i \succ_\ell s$ for all indices i in the range. We ignore from further computations those ranges for which we already know that $x_i \notin I$, and for the remaining ones repeat the procedure above with t instead of s . We end up with $\mathcal{O}(1)$ ranges of positions i for which $x_i \in I$. However, note that as the string sequence $(x_i)_{i=0}^k$ is always monotone (decreasing if $x_0 \preceq \rho^\infty$, increasing otherwise), these ranges (if any) can be merged into a single range, so in the output we end up with a single (possibly empty) periodic progression. \square

3.3.2 Definition of wavelet suffix trees

Let w be a string of length n . To define the wavelet suffix tree of w , we start from an auxiliary tree T of height $\mathcal{O}(\log n)$ with $\mathcal{O}(n \log n)$ nodes. Its leaves represent non-empty suffixes of w , and the left-to-right order of leaves corresponds to the lexicographic order on the suffixes. Internal nodes of T represent all substrings of w whose length is a power of two, with an exception of the root, which represents the empty word. Edges in T are defined so that a node representing v is an ancestor of a node representing v' if and only if v is a prefix of v' . To each non-root node u we assign a level $\ell(u) := 2|v|$, where v is the substring that u represents. For the root r , we set $\ell(r) := 1$. See Figure 4 for a sample tree T with levels assigned to nodes.

For a node u , we define $\mathcal{S}(u)$ to be the set of suffixes of w that are represented by descendants of u . Note that $\mathcal{S}(u)$ is a singleton if u is a leaf. The following observation characterizes the levels and sets $\mathcal{S}(u)$.

Observation 3.5. *For any node u other than the root:*

- (1) $\ell(\text{parent}(u)) \leq \ell(u)$,
- (2) if $y \in \mathcal{S}(u)$ and y' is a suffix of w such that $\text{lcp}(y, y') \geq \ell(\text{parent}(u))$, then $y' \in \mathcal{S}(u)$,
- (3) if $y, y' \in \mathcal{S}(u)$, then $\text{lcp}(y, y') \geq \lfloor \frac{1}{2} \ell(u) \rfloor$.

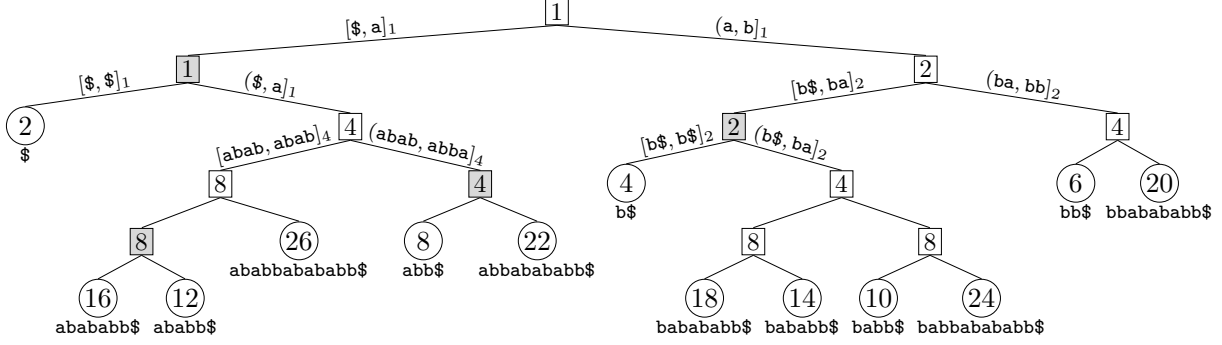


Figure 5: The wavelet suffix tree of $w = \text{ababbabababb}$ (see also Figures 3 and 4). Levels are written inside nodes. Gray nodes have been introduced as inner nodes of replacement trees. The corresponding suffix is written down below each leaf. Selected edges e are labelled with the intervals $I(e)$.

for adjacent edges e . The following lemma characterizes the intervals.

Lemma 3.6. *For any node u we have:*

- (1) *If u is not a leaf, then $I(u)$ is a disjoint union of $I(u, \text{lchild}(u))$ and $I(u, \text{rchild}(u))$.*
- (2) *If y is a suffix of w , then $y \in I(u)$ if and only if $y \in \mathcal{S}(u)$.*
- (3) *If u is not the root, then $I(u) \subseteq I(\text{parent}(u), u)$.*

Proof. (1) is a trivial consequence of the definitions.

(2) Clearly $y \in \mathcal{S}(u)$ iff $y \in [\min \mathcal{S}(u), \max \mathcal{S}(u)]$. Therefore, it suffices to show that if $\text{lcp}(y, y') \geq \ell(u)$ for $y' = \min \mathcal{S}(u)$ or $y' = \max \mathcal{S}(u)$, then $y \in \mathcal{S}(u)$. This is, however, a consequence of points (1) and (2) of Observation 3.5.

(3) Let $\ell_p = \ell(\text{parent}(u))$. If $u = \text{lchild}(\text{parent}(u))$, then $\mathcal{S}(u) \subseteq \mathcal{S}(\text{parent}(u))$ and, by Observation 3.5(1), $\ell(u) \leq \ell_p$, which implies the statement.

Therefore, assume that $u = \text{rchild}(\text{parent}(u))$, and let u' be the left sibling of u . Note that $I(\text{parent}(u), u) = (\max \mathcal{S}(u'), \max \mathcal{S}(u)]_{\ell_p}$ and $I(u) \subseteq [\min \mathcal{S}(u), \max \mathcal{S}(u)]_{\ell_p}$, since $\ell(u) \leq \ell_p$. Consequently, it suffices to prove that $\max \mathcal{S}(u') \prec_{\ell_p} \min \mathcal{S}(u)$. This is, however, a consequence of Observation 3.5(2) for $y = \min \mathcal{S}(u)$ and $y' = \max \mathcal{S}(u')$, and the fact that the left-to-right order of leaves coincides with the lexicographic order of the corresponding suffixes of w . \square

For each edge $e = (\text{parent}(u), u)$ of the wavelet suffix tree, we define $L(e)$ to be the sorted list of those substrings of w which belong to $I(e) \setminus I(u)$.

Recall that the wavelet suffix tree shall enjoy the *lexicographic property*: if we traverse the tree, and when going *down* an edge e we write out the contents of $L(e)$, whereas when visiting a leaf we output the corresponding suffix of w , we shall obtain a lexicographically sorted list of all substrings of w . This is proved in the following series of lemmas.

Lemma 3.7. *Let $e = (\text{parent}(u), u)$ for a node u . Substrings in $L(e)$ are smaller than any string in $I(u)$.*

Proof. We use a shorthand ℓ_p for $\ell(\text{parent}(u))$. Let $y = \max \mathcal{S}(u)$ be the rightmost suffix in the subtree of u . Consider a substring $s = w[k..j] \in L(e)$, also let $t = w[k..]$.$

We first prove that $s \preceq y$. Note that $I(e) = [x, y]_{\ell_p}$ or $I(e) = (x, y]_{\ell_p}$ for some string x . We have $s \in L(e) \subseteq I(e)$, and thus $s \preceq_{\ell_p} y$. If $\text{lcp}(s, y) < \ell_p$, this already implies that $s \preceq y$. Thus, let us assume that $\text{lcp}(s, y) \geq \ell_p$. The suffix t has s as a prefix, so this also means that $\text{lcp}(t, y) \geq \ell_p$. By Observation 3.5(2), $t \in \mathcal{S}(u)$, so $t \preceq y$. Thus $s \preceq t \preceq y$, as claimed.

To prove that $s \preceq y$ implies that s is smaller than any string in $I(u)$, it suffices to note that $y \in \mathcal{S}(u) \subseteq I(u)$, $s \notin I(u)$, and $I(u)$ is an interval. \square

Lemma 3.8. *The wavelet suffix tree satisfies the lexicographic property.*

Proof. Note that for the root r we have $I(r) = [\$, c]_1$ where c is the largest character present in w . Thus, $I(r)$ contains all substrings of $w\$$ and it suffices to show that if we traverse the subtree of r , writing out the contents of $L(e)$ when going down an edge e , and the corresponding suffix when visiting a leaf, we obtain a sorted list of substrings of $w\$$ contained in $I(r)$. But we will show even a stronger claim — we will show that, in fact, this property holds for all nodes u of the tree.

If u is a leaf this is clear, since $I(u)$ consists of the corresponding suffix of $w\$$ only. Next, if we have already proved the hypothesis for u , then prepending the output with the contents of $L(\text{parent}(u), u)$, by Lemmas 3.7 and 3.6(3), we obtain a sorted list of substrings of $w\$$ contained in $I(\text{parent}(u), u)$. Applying this property for both children of a non-leaf u' , we conclude that if the hypothesis holds for children of u' then, by Lemma 3.6(1), it also holds for u' . \square

Corollary 3.9. *Each list $L(e)$ contains consecutive prefixes of the largest element of $L(e)$.*

Proof. Note that if $x \prec y$ are substrings of w such that x is not a prefix of y , then x can be extended to a suffix x' of $w\$$ such that $x \prec x' \prec y$. However, $L(e)$ does not contain any suffix of $w\$$. By Lemma 3.8, $L(e)$ contains a consecutive collection of substrings of $w\$$, so x and y cannot be both present in $L(e)$. Consequently, each element of $L(e)$ is a prefix of $\max L(e)$.

Similarly, since $L(e)$ contains a consecutive collection of substrings of $w\$$, it must contain all prefixes of $\max L(e)$ no shorter than $\min L(e)$. \square

3.3.3 Implementation of auxiliary queries

Recall that $L_x(e)$ is the sublist of $L(e)$ containing suffixes of x . The wavelet suffix tree shall allow the following four types of queries in constant time:

- (1) For a substring x and an edge e , output the list $L_x(e)$ represented as $\mathcal{O}(1)$ non-overlapping periodic progressions;
- (2) Count the number of suffixes of $x = w[i..j]$ in the left/right subtree of a node (given along with the segment of its first bitmask corresponding to suffixes that start inside $[i, j]$);
- (3) Count the number of suffixes $x = w[i..j]$ that are preceded by a character c and lie in the left/right subtree of a node (given along with the segment of its second bitmask corresponding to suffixes that start inside $[i, j]$ and are preceded by c);
- (4) For a substring x and an edge e , compute the run-length encoding of the sequence of characters preceding suffixes in $L_x(e)$.

We start with an auxiliary lemma applied in the solutions to all four queries.

Lemma 3.10. *Let $e = (u, u')$ be an edge of a wavelet suffix tree of w , with u' being a child of u . The following operations can be implemented in constant time.*

- (1) Given a substring x of w , $|x| < \ell(u)$, return, as a single periodic progression of starting positions, all suffixes s of x such that $s \in I(e)$.
- (2) Given a range of positions $[i, j]$, $j - i \leq \ell(u)$, return all positions $k \in [i, j]$ such that $w[k..]\$ \in I(e)$, represented as at most two non-overlapping periodic progressions.

Proof. Let p be the longest common prefix of all strings in $I(u)$; by Observation 3.5(3), we have $|p| \geq \lfloor \frac{1}{2}\ell(u) \rfloor$.

(1) Assume $x = w[i..j]$. We apply Theorem 3.2 to find all occurrences of p within x , represented as a single periodic progression since $|x| + 1 < 2(|p| + 1)$. Then, using Lemma 3.4, we filter positions k for which $w[k, j] \in I(e)$.

(2) Let $x = w[i..j + |p| - 1]$ ($x = w[i..]\$$ if $j + |p| - 1 > |w|$). We apply Theorem 3.2 to find all occurrences of p within x , represented as at most two periodic progressions since $|x| + 1 \leq \ell(u) + |p| + 1 \leq 2\lfloor \frac{1}{2}\ell(u) \rfloor + |p| + 2 < 3(|p| + 1)$. Like previously, using Lemma 3.4 we filter positions k for which $w[k..]\$ \in I(e)$. \square

Lemma 3.11. *The wavelet suffix tree allows to answer queries (1) in constant time. In more details, for an edge $e = (\text{parent}(u), u)$, the starting positions of suffixes in $L_x(e)$ form at most three non-overlapping periodic progressions, which can be reported in $\mathcal{O}(1)$ time.*

Proof. First, we consider short suffixes. We use Lemma 3.10(1) to find all suffixes s of x , $|s| < \ell(\text{parent}(u))$, such that $s \in I(\text{parent}(u), u)$. Then, we apply Lemma 3.4 to filter out all suffixes belonging to $I(u)$. By Lemma 3.7, we obtain at most one periodic progression.

Now, it suffices to generate suffixes s , $|s| \geq \ell(\text{parent}(u))$, that belong to $L(e)$. Suppose $s = w[k..j]$. If $s \in I(e)$, then equivalently $w[k..]\$ \in I(e)$, since s is a long enough prefix of $w[k..]\$$ to determine whether the latter belongs to $I(e)$. Consequently, by Lemma 3.6, $w[k..]\$ \in I(u)$. This implies $|s| < \ell$ (otherwise we would have $s \in I(u)$), i.e., $k \in [j - \ell + 2, j - \ell(\text{parent}(u)) + 1]$. We apply Lemma 3.10(2) to compute all positions k in this range for which $w[k..]\$ \in I(e)$. Then, using Lemma 3.4, we filter out positions k such that $w[k..j] \in I(u)$. By Lemma 3.7, this cannot increase the number of periodic progressions, so we end up with three non-overlapping periodic progressions in total. \square

Lemma 3.12. *The wavelet suffix tree allows to answer queries (2) in constant time.*

Proof. Let u be the given node and u' be its right/left child (depending on the variant of the query). First, we use Lemma 3.10(1) to find all suffixes s of x , $|s| < \ell(u)$, such that $s \in I(u, u')$, i.e., s lies in the appropriate subtree of u .

Thus, it remains to count suffixes of length at least $\ell(u)$. Suppose $s = w[k..j]$ is a suffix of x such that $|s| \geq \ell(u)$ and $s \in I(u, u')$. Then $w[k..]\$ \in I(u, u')$, and the number of suffixes $w[k..]\$ \in I(u, u')$ such that $k \in [i, j]$ is simply the number of 1's or 0's in the given segment of the first bitmask in u , which we can compute in constant time. Observe, however, that we have also counted positions k such that $|w[k..j]| < \ell(u)$, and we need to subtract the number of these positions. For this, we use Lemma 3.10(2) to compute the positions $k \in [j - \ell + 2, j]$ such that $w[k..]\$ \in I(u, u')$. We count the total size of the obtained periodic progressions, and subtract it from the final result, as described. \square

Lemma 3.13. *The wavelet suffix tree allows to answer queries (3) and (4) in $\mathcal{O}(1)$ time.*

Proof. Observe that for any periodic progression $p_0 \dots, p_k$ we have $w[p_1 - 1] = \dots = w[p_k - 1]$. Thus, it is straightforward to determine which positions of such a progression are preceded by c .

Answering queries (3) is analogous to answering queries (2), we just use the second bitmask at the given node and consider only positions preceded by c while counting the sizes of periodic progressions.

To answer queries (4), it suffices to use Lemma 3.11 to obtain $L_x(e)$. By Corollary 3.9, suffixes in $L_x(e)$ are prefixes of one another, so the lexicographic order on these suffixes coincides with the order of ascending lengths. Consequently, the run-length encoding of the piece corresponding to $L_x(e)$ has at most six phrases and can be easily found in $\mathcal{O}(1)$ time. \square

3.4 Construction of wavelet suffix trees

The actual construction algorithm is presented in Section 3.4.2. Before, in Section 3.4.1, we introduce several auxiliary tools for abstract weighted trees.

3.4.1 Toolbox for weighted trees

Let T be a rooted ordered tree with positive integer weights on edges, n leaves and no inner nodes of degree one. We say that L_1, \dots, L_{n-1} is an *LCA sequence* of T , if L_i is the (weighted) depth of the lowest common ancestor of the i -th and $(i+1)$ -th leaves. The following fact is usually applied to construct the suffix tree of a string from the suffix array and the LCP table [12].

Fact 3.14. *Given a sequence $(L_i)_{i=1}^{n-1}$ of non-negative integers, one can construct in $\mathcal{O}(n)$ time a tree whose LCA sequence is $(L_i)_{i=1}^{n-1}$.*

The LCA sequence suffices to detect if a tree is binary.

Observation 3.15. *A tree is a binary tree if and only if its LCA sequence $(L_i)_{i=1}^{n-1}$ satisfies the following property for every $i < j$: if $L_i = L_j$ then there exists k , $i < k < j$, such that $L_k < L_i$.*

Trees constructed by the following lemma can be seen as a variant of the weight-balanced trees, whose existence for arbitrary weights was proved by Blum and Mehlhorn [5].

Lemma 3.16. *Given a sequence w_1, \dots, w_n of positive integers, one can construct in $\mathcal{O}(n)$ time a binary tree T with n leaves, such that the depth of the i -th leaf is $\mathcal{O}(1 + \log \frac{W}{w_i})$, where $W = \sum_{j=1}^n w_j$.*

Proof. For $i = 0, \dots, n$ define $W_i = \sum_{j=1}^i w_j$. Let p_i be the position of the most significant bit where the binary representations of W_{i-1} and W_i differ, and let $P = \max_{i=1}^n p_i$. Observe that $P = \Theta(\log W)$ and $p_i = \Omega(\log w_i)$. Using Fact 3.14, we construct a tree T whose LCA sequence is $L_i = P - p_i$. Note that this sequence satisfies the condition of Observation 3.15, and thus the tree is binary.

Next, we insert an extra leaf between the two children of any node to make the tree ternary. The i -th of these leaves is inserted at (weighted) depth $1 + L_i = \mathcal{O}(1 + \log W - \log w_i)$, which is also an upper bound for its unweighted depth. Next, we remove the original leaves. This way we get a tree satisfying the lemma, except for the fact that inner nodes may have between one and three children, rather than exactly two. In order to resolve this issue, we remove (dissolve) all inner nodes with exactly one child, and for each node u with three children v_1, v_2, v_3 , we introduce a new node u' , setting v_1, v_2 as the children of u' and u', v_3 as the children of u . This way we get a full binary tree, and the depth of any node may increase at most twice, i.e., for the i -th leaf it stays $\mathcal{O}(1 + \log \frac{W}{w_i})$. \square

Let T be an ordered rooted tree and let u be a node of T , which is neither the root nor a leaf. Also, let v be the parent of u . We say that T' is obtained from T by *contracting* the edge (v, u) , if u is removed and the children of u replace u at its original location in the list of children of v . If T' is obtained from T by a sequence of edge contractions, we say that T' is a *contraction* of T . Note that contraction does not alter the pre-order and post-order of the preserved nodes, which implies that the ancestor-descendant relation also remains unchanged for these nodes.

Corollary 3.17. *Let T be an ordered rooted tree of height h , which has n leaves and no inner node with exactly one child. Then, in $\mathcal{O}(n)$ time one can construct a full binary ordered rooted tree T' of height $\mathcal{O}(h + \log n)$ such that T is a contraction of T' and T' has $\mathcal{O}(n)$ nodes.*

Proof. For any node u of T with three or more children, we replace the star-shaped tree joining it with its children v_1, \dots, v_k by an appropriate replacement tree. Let $W(u)$ be the number of leaves in the subtree of u , and let $W(v_i)$ be the number of leaves in the subtrees below v_i , $1 \leq i \leq k$. We use Lemma 3.16 for $w_i = W(v_i)$ to construct the replacement tree. Consequently, a node u with depth d in T has depth $\mathcal{O}(d + \log \frac{n}{W(u)})$ in T' (easy top-down induction). The resulting tree has height $\mathcal{O}(h + \log n)$, as claimed. \square

3.4.2 The algorithm

In this section we show how to construct the wavelet suffix tree of a string w of length n in $\mathcal{O}(n\sqrt{\log n})$ time. The algorithm is deterministic, but the data structure of Theorem 3.2, required by the wavelet suffix tree, has a randomized construction only, running in $\mathcal{O}(n)$ expected time.

The construction algorithm has two phases: first, it builds the *shape* of the wavelet suffix tree, following a description in Section 3.3.2, and then it uses the results of Section 2 to obtain the bitmasks.

We start by constructing the suffix array and the LCP table for $w\$$ (see [12]). Under the assumption that $\sigma < n$, this takes linear time.

Recall that in the definition of the wavelet suffix tree we started with a tree of size $\mathcal{O}(n \log n)$. For an $\mathcal{O}(n \log n)$ -time construction we cannot afford that. Thus, we construct the tree T already without inner nodes having exactly one child. Observe that this tree is closely related to the suffix tree of $w\$$. The only difference is that if the longest common prefix of two consecutive suffixes is d , their root-to-leaf paths diverge at depth $\lfloor \log d \rfloor$ instead of d . To overcome this difficulty, we use Fact 3.14 for $L_i = \lfloor \log LCP[i] \rfloor$, rather than $LCP[i]$ which we would use for the suffix tree. This way an inner node u at depth j represents a substring of length 2^j . The level $\ell(u)$ of an inner node u is set to 2^{j+1} , and if u is a leaf representing a suffix s of $w\$$, we have $\ell(u) = 2|s|$.

After this operation, the tree T may have inner nodes of large degree, so we use Corollary 3.17 to obtain a binary tree such that T is its contraction. We set this binary tree as the shape of the wavelet suffix tree. Since T has height $\mathcal{O}(\log n)$, so does the wavelet suffix tree.

To construct the bitmasks, we apply Theorem 2.4 for T with the leaf representing $w[i..]\$$ assigned to i . For the first bitmask, we simply set $s[i] = i$. For the second bitmask, we sort all positions i with respect to $(w[i-1], i)$ and take the resulting sequence as s .

This way, we complete the proof of the main theorem concerning wavelet suffix trees.

Theorem 3.18. *A wavelet suffix tree of a string w of length n occupies $\mathcal{O}(n)$ space and can be constructed in $\mathcal{O}(n\sqrt{\log n})$ expected time.*

3.5 Applications

3.5.1 Substring suffix rank/select

In the substring suffix rank problem, we are asked to find the rank of a substring y among the suffixes of another substring x . The substring suffix selection problem, in contrast, is to find the k -th lexicographically smallest suffix of x for a given an integer k and a substring x of w .

Theorem 3.19. *The wavelet suffix tree can solve the substring suffix rank problem in $\mathcal{O}(\log n)$ time.*

Proof. Using binary search on the leaves of the wavelet suffix tree of w , we locate the minimal suffix t of $w\$$ such that $t \succ y$. Let π denote the path from the root to the leaf corresponding to t . Due to the lexicographic property, the rank of y among the suffixes of x is equal to the sum of two numbers. The first one is the number of suffixes of x in the left subtrees hanging from the path π , whereas the second summand is the number of suffixes not exceeding y in the lists $L_x(e)$ for $e \in \pi$.

To compute those two numbers, we traverse π maintaining a segment $[\ell, r]$ of the first bitmask corresponding to the suffixes of $w\$$ starting within x . When we descend to the left child, we set $[\ell, r] := [\text{rank}_0(\ell), \text{rank}_0(r)]$, while for the right child, we set $[\ell, r] := [\text{rank}_1(\ell), \text{rank}_1(r)]$. In the latter case, we pass $[\ell, r]$ to type (2) queries, which let us count the suffixes of x in the left subtree hanging from π in the current node. This way, we compute the first summand.

For the second number, we use type (1) queries to generate all lists $L_x(e)$ for $e \in \pi$. Note that if we concatenated these lists $L_x(e)$ in the root-to-leaf order of edges, we would obtain a sorted list of strings. Thus, while processing the lists in this order (ignoring the empty ones), we add up the sizes of $L_x(e)$ until $\max L_x(e) \succ y$. For the first encountered list $L_x(e)$ satisfying this property, we binary search within $L_x(e)$ to determine the number of elements not exceeding y , and also add this value to the final result.

The described procedure requires $\mathcal{O}(\log n)$ time, since type (1) and (2) queries, as well as substring comparison queries (Lemma 3.1), run in $\mathcal{O}(1)$ time. \square

Theorem 3.20. *The wavelet suffix tree can solve the substring suffix selection problem in $\mathcal{O}(\log n)$ time.*

Proof. The algorithm traverses a path in the wavelet suffix tree of w . It maintains a segment $[\ell, r]$ of the first bitmask corresponding to suffixes of w starting within $x = w[i..j]$, and a variable k' counting the suffixes of x represented in the left subtrees hanging from the path on the edges of the path. The algorithm starts at the root initializing $[\ell, r]$ with $[i, j]$ and $k' = 0$.

At each node u , it first decides to which child of u to proceed. For this, it performs a type (2) query to determine k'' , the number of suffixes of x in the left subtree of u . If $k' + k'' \geq k$, it chooses to go to the left child, otherwise to the right one; in the latter case it also updates $k' := k' + k''$. The algorithm additionally updates the segment $[\ell, r]$ using the rank queries on the bitmask.

Let u' be the child of u that the algorithm has chosen to proceed to. Before reaching u' , the algorithm performs a type (1) query to compute $L_x(u, u')$. If k' summed with the size of this list is at least k , then the algorithm terminates, returning the $k - k'$ -th element of the list (which is easy to retrieve from the representation as a periodic progression). Otherwise, it sets $k' := k' + |L_x(u, u')|$, so that k' satisfies the definition for the extended path from the root to u' .

The correctness of the algorithm follows from the lexicographic property, which implies that at the beginning of each step, the sought suffix of x is the $k - k'$ -th smallest suffix in the subtree

of u . In particular, the procedure always terminates before reaching a leaf. The running time of the algorithm is $\mathcal{O}(\log n)$ due to $\mathcal{O}(1)$ -time implementations of type (1) and (2) queries. \square

We now show that the query time for the two considered problems is almost optimal. We start by reminding lower bounds by Pătraşcu and by Jørgensen and Larsen.

Theorem 3.21 ([30, 29]). *In the cell-probe model with W -bit cells, a static data structure of size $c \cdot n$ must take $\Omega(\frac{\log n}{\log c + \log W})$ time for orthogonal range counting queries.*

Theorem 3.22 ([22]). *In the cell-probe model with W -bit cells, a static data structure of size $c \cdot n$ must take $\Omega(\frac{\log n}{\log c + \log W})$ time for orthogonal range selection queries.*

Both of these results allow the coordinates of points to be in the *rank space*, i.e., for each $i \in \{1, \dots, n\}$ there is one point $(i, A[i])$, and values $A[i]$ are distinct integers in $\{1, \dots, n\}$.

This lets us construct a string $w = A[1] \dots A[n]$ for any given point set P . Since w has pairwise distinct characters, comparing suffixes of any substring of w is equivalent to comparing their first characters. Consequently, the substring suffix selection in w is equivalent to the orthogonal range selection in P , and the substring suffix rank in w is equivalent to the orthogonal range counting in P (we need to subtract the results of two suffix rank queries to answer an orthogonal range counting query). Consequently, we obtain

Corollary 3.23. *In the cell-probe model with W -bit cells, a static data structure of size $c \cdot n$ must take $\Omega(\frac{\log n}{\log c + \log W})$ time for the substring suffix rank and the substring suffix select queries.*

3.5.2 Run-length encoding of the BWT of a substring

Wavelet suffix trees can be also used to compute the run-length encoding of the Burrows-Wheeler transform of a substring. We start by reminding the definitions. The *Burrows-Wheeler transform* [7] (BWT) of a string x is a string $b_0 b_1 \dots b_{|x|}$, where b_k is the character preceding the k -th lexicographically smallest suffix of $x\$$. The BWT tends to contain long segments of equal characters, called *runs*. This, combined with *run-length encoding*, allows to compress strings efficiently. The *run-length encoding* of a string is obtained by replacing each maximal run by a pair: the character that forms the run and the length of the run. For example, the BWT of a string *banana* is *annb\$aa*, and the run-length encoding of *annb\$aa* is *a1n2b1\$1a2*.

Below, we show how to compute the run-length encoding of the BWT of a substring $x = w[i..j]$ using the wavelet suffix tree of w . Let $x = w[i..j]$ and for $k \in \{1, \dots, |x|\}$ let $s_k = w[i_k..j]$ be the suffixes of x sorted in the lexicographic order. Then the BWT of x is equal to $b_0 b_1 \dots b_{|x|}$, where $b_0 = w[j]$, and for $k \geq 1$, $b_k = w[i_k - 1]$, unless $i_k = i$ when $b_k = \$$.

Our algorithm initially generates a string $b(x)$ which instead of $\$$ contains $w[i - 1]$. However, we know that $\$$ should occur at the position equal to the rank of x among all the suffixes of x . Consequently, a single substring suffix rank query suffices to find the position which needs to be corrected.

Remember that the wavelet suffix tree satisfies the lexicographic property. Consequently, if we traverse the tree and write out the characters preceding the suffixes in the lists $L_x(e)$, we obtain $b(x)$ (without the first symbol b_0). Our algorithm simulates such a traversal. Assume that the last character appended to $b(x)$ is c , and the algorithm is to move down an edge $e = (u, u')$. Before deciding to do so, it checks whether all the suffixes of x in the appropriate (left or right) subtree of u are preceded with c . For this, it performs type (2) and (3) queries, and if both results are

equal to some value q , it simply appends c^q to $b(x)$ and decides not to proceed to u' . In order to make these queries possible, for each node on the path from the root to u , the algorithm maintains segments corresponding to $[i, j]$ in the first bitmasks, and to $(c, [i, j])$ in the second bitmasks. These segments are computed using rank queries on the bitmasks while moving down the tree.

Before the algorithm continues at u' , if it decides to do so, suffixes in $L_x(e)$ need to be handled. We perform a type (4) query to compute the characters preceding these suffixes, and append the result to $b(x)$. This, however, may result in c no longer being the last symbol appended to $b(x)$. If so, the algorithm updates the segments of the second bitmask for all nodes on the path from the root to u' . We assume that the root stores all positions i sorted by $(w[i - 1], i)$, which lets us use a binary search to find either endpoint of the segment for the root. For the subsequent nodes on the path, the rank structures on the second bitmasks are applied. Overall, this update takes $\mathcal{O}(\log n)$ time and it is necessary at most once per run.

Now, let us estimate the number of edges visited. Observe that if we go down an edge, then the last character of $b(x)$ changes before we go up this edge. Thus, all the edges traversed down between such character changes form a path. The length of any path is $\mathcal{O}(\log n)$, and consequently the total number of visited edges is $\mathcal{O}(s \log n)$, where s is the number of runs.

Theorem 3.24. *The wavelet suffix tree can compute the run-length encoding of the BWT of a substring x in $\mathcal{O}(s \log n)$ time, where s is the size of the encoding.*

3.5.3 Speeding up queries

Finally, we note that building wavelet suffix trees for several substrings of w , we can make the query time adaptive to the length of the query substring x , i.e., replace $\mathcal{O}(\log n)$ by $\mathcal{O}(\log |x|)$.

Theorem 3.25. *Using a data structure of size $\mathcal{O}(n)$, which can be constructed in $\mathcal{O}(n\sqrt{\log n})$ expected time, substring suffix rank and selection problems can be solved in $\mathcal{O}(\log |x|)$ time. The run-length encoding $b(x)$ of the BWT of a substring x can be found in $\mathcal{O}(|b(x)| \log |x|)$ time.*

Proof. We build wavelet suffix trees for some substrings of length $n_k = \lfloor n^{2^{-k}} \rfloor$, $0 \leq k \leq \log \log n$. For each length n_k we choose every $\lfloor \frac{1}{2}n_k \rfloor$ -th substring, starting from the prefix and, additionally, we choose the suffix. Auxiliary data structures of Lemma 3.1 and Theorem 3.2, are built for w only.

We have $n_k = \lfloor \sqrt{n_{k-1}} \rfloor$, so $n_{k-1} \leq (n_k + 1)^2$ and thus any substring x of w lies within a substring v , $|v| \leq 2(|x| + 1)^2$, for which we store the wavelet suffix tree. For each m , $1 \leq m \leq n$, we store such n_k that $2m \leq n_k \leq 2(m + 1)^2$. This reduces finding an appropriate substring v to simple arithmetics. Using the wavelet suffix tree for v instead of the tree for the whole string w gives the announced query times. The only thing we must be careful about is that the input for the substring suffix rank problem also consists of a string y , which does not need to be a substring of v . However, looking at the query algorithm, it is easy to see that y is only used through the data structure of Lemma 3.1.

It remains to analyze the space usage and construction time. Observe that the wavelet suffix tree of a substring v is simply a binary tree with two bitmasks at each node and with some pointers to the positions of the string w . In particular, it does not contain any characters of w and, if all pointers are stored as relative values, it can be stored using $\mathcal{O}(|v| \log |v|)$ bits, i.e., $\mathcal{O}(|v| \frac{\log |v|}{\log n})$ words. For each n_k the total length of selected substrings is $\mathcal{O}(n)$, and thus the space usage is $\mathcal{O}(n \frac{\log n_k}{\log n}) = \mathcal{O}(n 2^{-k})$, which sums up to $\mathcal{O}(n)$ over all lengths n_k . The construction

time is $\mathcal{O}(|v|\sqrt{\log |v|})$ for any substring (including alphabet renumbering), and this sums up to $\mathcal{O}(n\sqrt{2^{-k} \log n})$ for each length, and $\mathcal{O}(n\sqrt{\log n})$ in total. \square

Acknowledgement

The authors gratefully thank Djamel Belazzougui, Travis Gagie, and Simon J. Puglisi who pointed out that wavelet suffix trees can be useful for BWT-related applications.

References

- [1] Amihood Amir, Alberto Apostolico, Gad M. Landau, Avivit Levy, Moshe Lewenstein, and Ely Porat. Range LCP. *Journal of Computer and System Sciences*, 80(7):1245–1253, 2014.
- [2] Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms*, 3(2), May 2007.
- [3] Maxim Babenko, Paweł Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Computing minimal and maximal suffixes of a substring revisited. In Alexander S. Kulikov, Sergei O. Kuznetsov, and Pavel Pevzner, editors, *Combinatorial Pattern Matching*, volume 8486 of *LNCS*, pages 30–39. Springer International Publishing, 2014.
- [4] Maxim Babenko, Ignat Kolesnichenko, and Tatiana Starikovskaya. On minimal and maximal suffixes of a substring. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching*, volume 7922 of *LNCS*, pages 28–37. Springer Berlin Heidelberg, 2013.
- [5] Norbert Blum and Kurt Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11(3):303–320, July 1980.
- [6] Gerth Stølting Brodal, Beat Gfeller, Allan Grønlund Jørgensen, and Peter Sanders. Towards optimal range median. *Theoretical Computer Science: Selected Papers from 36th International Colloquium on Automata, Languages and Programming (ICALP 2009)*, 412(24):2588–2601, 2011.
- [7] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.
- [8] Timothy M. Chan and Mihai Pătrașcu. Counting inversions, offline orthogonal range counting, and related problems. In *21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA’10*, pages 161–173. SIAM, 2010.
- [9] David Richard Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1996.
- [10] Francisco Claude, Patrick K. Nicholson, and Diego Seco. Space efficient wavelet tree construction. In Roberto Grossi, Fabrizio Sebastiani, and Fabrizio Silvestri, editors, *String Processing and Information Retrieval*, volume 7024 of *LNCS*, pages 185–196. Springer Berlin Heidelberg, 2011.

- [11] Graham Cormode and S. Muthukrishnan. Substring compression problems. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'05, pages 321–330. SIAM, 2005.
- [12] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, Cambridge, 2007.
- [13] Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word from its runs structure. *Theoretical Computer Science*, 521:29–41, 2014.
- [14] Jean-Pierre Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4(4):363–381, 1983.
- [15] Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In Dan Hirschberg and Gene Myers, editors, *Combinatorial Pattern Matching*, volume 1075 of *LNCS*, pages 130–140. Springer Berlin Heidelberg, 1996.
- [16] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), May 2007.
- [17] Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, October 2006.
- [18] Paweł Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. Weighted ancestors in suffix trees. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms – ESA 2014*, volume 8737 of *LNCS*, pages 455–466. Springer Berlin Heidelberg, 2014.
- [19] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'03, pages 841–850. SIAM, 2003.
- [20] Roberto Grossi and Giuseppe Ottaviano. The wavelet trie: maintaining an indexed sequence of strings in compressed space. In *31st Symposium on Principles of Database Systems*, PODS'12, pages 203–214. ACM, 2012.
- [21] Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554. IEEE Computer Society, 1989.
- [22] Allan Grønlund Jørgensen and Kasper Green Larsen. Range selection and median: tight cell probe lower bounds and adaptive data structures. In *22nd Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'11, pages 805–813. SIAM, 2011.
- [23] Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science: Advances in Stringology*, 525:42–54, 2014.

- [24] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient data structures for the factor periodicity problem. In Liliana Calderin-Benavides, Cristina Gonzalez-Caro, Edgar Chavez, and Nivio Ziviani, editors, *String Processing and Information Retrieval*, volume 7608 of *LNCS*, pages 284–294. Springer Berlin Heidelberg, 2012.
- [25] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *26th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA’15. SIAM, 2015.
- [26] Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.
- [27] Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms: 23rd Annual Symposium on Combinatorial Pattern Matching*, 25:2–20, 2014.
- [28] Manish Patil, Rahul Shah, and Sharma V. Thankachan. Faster range LCP queries. In Oren Kurland, Moshe Lewenstein, and Ely Porat, editors, *String Processing and Information Retrieval*, volume 8214 of *LNCS*, pages 263–270. Springer International Publishing, 2013.
- [29] Mihai Pătraşcu. Lower bounds for 2-dimensional range counting. In *39th Annual ACM Symposium on Theory of Computing*, STOC’07, pages 40–46. ACM, 2007.
- [30] Mihai Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM Journal on Computing*, 40(3):827–847, 2011.
- [31] Milan Ružić. Constructing efficient dictionaries in close to sorting time. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfssdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming – ICALP 2008*, volume 5125 of *LNCS*, pages 84–95, 2008.
- [32] German Tischler. On wavelet tree construction. In Raffaele Giancarlo and Giovanni Manzini, editors, *Combinatorial Pattern Matching*, volume 6661 of *LNCS*, pages 208–218. Springer Berlin Heidelberg, 2011.

A Constructing rank/select structures

Lemma 2.3. *Given a bit string $B[1..N]$ packed in $\frac{N}{\log n}$ machine words, we can extend it in $\mathcal{O}(\frac{N}{\log n})$ time with a rank/select data structure occupying $o(\frac{N}{\log n})$ additional space, assuming $\tilde{\mathcal{O}}(\sqrt{n})$ time and space preprocessing shared by all instances of the structure.*

Proof. We focus on implementing rank_1 and select_1 . By repeating the construction, we also get rank_0 and select_0 .

Rank. Recall the usual implementation of rank_1 . We first split B into superblocks of length $\log^2 n$, and then split every superblock into blocks of length $\frac{1}{2} \log n$. We store the cumulative rank for each superblock, and the cumulative rank within the superblock for each block. All ranks can be computed by reading whole blocks. More precisely, in every step we extract the next $\frac{1}{2} \log n$ bits, i.e., take either the lower or the higher half of the next word encoding the input, and

compute the number of ones inside using a shared table of size $\mathcal{O}(\sqrt{n})$. Hence we need $\mathcal{O}(\frac{N}{\log n})$ -time preprocessing for rank queries.

Select. Now, recall the more involved implementation of select_1 . We split B into superblocks by choosing every s -th occurrence of $\mathbf{1}$, where $s = \log n \log \log n$, and store the starting position of every superblock together with a pointer to its description, using $\mathcal{O}(\frac{N}{s} \log n) = o(N)$ bits in total. This can be done by processing the input word-by-word while maintaining the total number of $\mathbf{1}$ s seen so far in the current superblock. After reading the next word, i.e., the next $\log n$ bits, we check if the next superblock should start inside, and if so, compute its starting position. This can be easily done if we can count $\mathbf{1}$ s inside the word and select the k -th one in $\mathcal{O}(1)$ time, which can be preprocessed in a shared table of size $\mathcal{O}(\sqrt{n})$ if we split every word into two parts. There are at most $\frac{N}{s}$ superblocks, so the total construction time so far is $\mathcal{O}(\frac{N}{\log n} + \frac{N}{s}) = \mathcal{O}(\frac{N}{\log n})$. Then we have two cases depending on the length ℓ of the superblock.

If $\ell > s^2$, we store the position of every $\mathbf{1}$ inside the superblock explicitly. We generate the positions by sweeping the superblock word-by-word, and extracting the $\mathbf{1}$ s one-by-one, which takes $\frac{\ell}{\log n} + s$ time. Because there are at most $\frac{N}{s^2}$ such sparse superblocks, this is $\mathcal{O}(\frac{N}{\log n} + \frac{N}{s}) = \mathcal{O}(\frac{N}{\log n})$, and the total number of used bits is $\mathcal{O}(\frac{N}{s^2} s \log n) = o(N)$.

If $\ell \leq s^2$, the standard solution is to recurse on the superblock, i.e., repeat the above with n replaced by s^2 . We need to be more careful, because otherwise we would have to pay $\mathcal{O}(1)$ time for roughly every $(\log \log n)^2$ -th occurrence of $\mathbf{1}$ in B , which would be too much.

We want to split every superblock into blocks by choosing every s' -th occurrence of $\mathbf{1}$, where $s' = (\log \log n)^2$, and storing the starting position of every block relative to the starting position of its superblock. Then, if a block is of length $\ell' > s'^2$, we store the position of every $\mathbf{1}$ inside the block, again relative to the starting position of the superblock, which takes $\mathcal{O}(\frac{N}{s'^2} s' \log(s^2)) = o(N)$ bits in total. Otherwise, the block is so short that we can extract the k -th occurrence of $\mathbf{1}$ inside using the shared table. The starting positions of the blocks are saved one after another using $\mathcal{O}(\frac{N}{s'} \log(s^2)) = o(N)$ bits in total, so that we can access the i -th block in $\mathcal{O}(1)$ time. With the starting positions of every $\mathbf{1}$ in a sparse block the situation is slightly more complicated, as not every block is sparse, and we cannot afford to store a separate pointer for every block. The descriptions of all sparse blocks are also saved one after another, but additionally we store for every block a single bit denoting whether it is sparse or dense. These bits are concatenated together and enriched with a rank_1 structure, which allows us to compute in $\mathcal{O}(1)$ time where the description of a given sparse block begins.

To partition a superblock into blocks efficiently, we must be able to generate multiple blocks simultaneously. Assume that a superblock spans a number of whole words (if not, we can shift its description paying $\mathcal{O}(1)$ per word, which is fine). Then we can process these words one-by-one after some preprocessing. First, consider the process of creating the blocks when we read the bits one-by-one. We maintain the description of the current block, which consists of its length $\ell' \leq \ell$, the number of $\mathbf{1}$ s seen so far, and their starting positions. The total size of the description is $\mathcal{O}(\log \ell + s' \log \ell)$ bits. Additionally, we maintain our current position inside the superblock, which takes another $\mathcal{O}(\log \ell)$ bits. After reading the next bit, we update the current position, and either update the description of the current block, or start a new block. In the latter case, we output the position of the first $\mathbf{1}$ inside the current block, store one bit denoting whether $\ell' > s'^2$, and if so, additionally save the positions of all $\mathbf{1}$ s inside the current block.

Now we want to accelerate processing the words describing a superblock. Informally, we would

like to read the whole next word, and generate all the corresponding data in $\mathcal{O}(1)$ time. The difficulty is that the starting positions of the blocks, the bits denoting whether a block is sparse or dense, and the descriptions of sparse blocks are all stored in separate locations. To overcome this issue, we can view the whole process as an automaton with one input tape, three separate output tapes, and a state described by $\mathcal{O}(\log \ell + s' \log \ell) = \mathcal{O}(\text{polyloglog}(n))$ bits. In every step, the automaton reads the next bit from the input tape, updates its state, and possibly writes into the output tapes. Because the situation is fully deterministic, we can preprocess its behavior, i.e., for every initial state and a sequence of $\frac{1}{2} \log n$ bits given in the input tape, we can precompute the new state and the data written into the output tapes (observe that, by construction, this is always at most $\log n$ bits). The cost of such preprocessing is $\mathcal{O}(2^{\frac{1}{2} \log n + \text{polyloglog}(n)}) = \tilde{\mathcal{O}}(\sqrt{n})$. The output buffers are implemented as packed lists, so appending at most $\log n$ bits to any of them takes just $\mathcal{O}(1)$ time. Therefore, we can process $\frac{1}{2} \log n$ bits in $\mathcal{O}(1)$ time, which accelerates generating the blocks by the desired factor of $\log n$. \square

Lemma 2.6. *Let $d \leq \log^\varepsilon n$ for $\varepsilon < \frac{1}{3}$. Given a string $D[1..N]$ over the alphabet $[0, d-1]$ packed in $\frac{N \log d}{\log n}$ machine words, we can extend it in $\mathcal{O}(\frac{N \log d}{\log n})$ time with a generalized rank/select structure occupying additional $o(\frac{N}{\log n})$ space, assuming an $\tilde{\mathcal{O}}(\sqrt{n})$ time and space preprocessing shared by all instances of the structure.*

Proof. The construction is similar to the one from Lemma 2.3, but requires careful adjustment of the parameters. The main difficulty is that we cannot afford to consider every $c \in [0, d-1]$ separately as in the binary case.

Rank. We split D into superblocks of length $d \log^2 n$, and then split every superblock into blocks of length $\frac{1}{3} \frac{\log n}{\log d}$. For every superblock, we store the cumulative generalized rank of every character, i.e., for every character c we store the number of positions where characters $c' \leq c$ occur in the prefix of the string up to the beginning of the superblock. This takes $\mathcal{O}(d \log n)$ bits per superblock, which is $\mathcal{O}(\frac{N}{\log n}) = o(N)$ in total.

For every block, we store the cumulative generalized rank of every character within the superblock, i.e., for every character c we store the number of positions where characters $c' \leq c$ occur in the prefix of the superblock up to the beginning of the block. This takes $\mathcal{O}(d \log(d \log^2 n)) = \mathcal{O}(\log^\varepsilon n \log \log n)$ bits per block, which is $\mathcal{O}(N / \frac{\log n}{\log d} \log^\varepsilon n \log \log n) = o(N)$ in total.

Finally, we store a shared table, which given a string of $\frac{1}{3} \frac{\log n}{\log d}$ characters packed into a single machine word of length $\frac{1}{3} \log n$, a character c , and a number i , returns the number of positions where characters $c' \leq c$ occur in the prefix of length i . Both the size and the construction time of the table is $\tilde{\mathcal{O}}(2^{\frac{1}{3} \log n})$. It is clear that using the table, together with the cumulative ranks within the blocks and superblocks, we can compute any generalized rank in $\mathcal{O}(1)$ time.

We proceed with a construction algorithm. All ranks can be computed by reading whole blocks. More precisely, the cumulative generalized ranks of the next block can be computed by taking the cumulative generalized ranks of the previous block (if it belongs to the same superblock) stored in one word of length $d \log(d \log^2 n)$, and updating it with the next $\frac{1}{3} \frac{\log n}{\log d}$ characters stored in one word of length $\frac{1}{3} \log n$. Such updates can be preprocessed for every possible input in $\tilde{\mathcal{O}}(2^{\log^\varepsilon n + \frac{1}{3} \log n})$ time and space. Then each block can be handled in $\mathcal{O}(1)$ time, which gives $\mathcal{O}(\frac{N \log d}{\log n})$ in total.

The cumulative generalized ranks of the next superblock can be computed by taking the cumulative generalized ranks of the previous superblock, and updating it with the cumulative gen-

eralized ranks of the last block in that previous superblock. This can be done by simply iterating over the whole alphabet and updating each generalized rank separately in $\mathcal{O}(1)$ time, which is $\mathcal{O}(\frac{N}{d \log^2 n} d) = o(\frac{N \log d}{\log n})$ in total.

Select. We store a separate structure for every $c \in [0, d-1]$. Even though the structures must be constructed together to guarantee the promised complexity, first we describe a single such structure. Nevertheless, when stating the total space usage, we mean the sum over all $c \in [0, d-1]$.

We split D into superblocks by choosing every s -th occurrence of c in D , where $s = d \log^2 n$, and store the starting position of every superblock. This takes $\mathcal{O}((d + \frac{N}{s}) \log n) = o(N)$ bits in total. As in the binary case, then we proceed differently depending on the length ℓ of the superblock.

If $\ell > s^2$, we store the position of every occurrence of c inside the superblock explicitly. This takes $\mathcal{O}(d \frac{N}{s^2} s \log n) = o(N)$ bits.

If $\ell \leq s^2$, we split the superblock into smaller blocks by choosing every s' -th occurrence of c inside, where $s' = d(\log \log n)^2$. We write down the starting position of every block relative to the starting position of the superblock, which takes $\mathcal{O}((d + \frac{N}{s'}) \log s) = \mathcal{O}(\frac{N}{d \log \log n}) = o(\frac{N}{d})$ bits in total. Then, if the block is of length $\ell' > d \cdot s'^2$, we store the position of every occurrence of c inside, again relative to the starting position of the superblock, which takes $\mathcal{O}(d \frac{N}{d \cdot s'^2} s' \log s) = \mathcal{O}(\frac{N}{d \log \log n}) = o(\frac{N}{d})$ bits in total. Otherwise, the block spans at most $\mathcal{O}(\log^{3\varepsilon} n (\log \log n)^4) = o(\log n)$ bits in D , hence the k -th occurrence of c there can be extracted in $\mathcal{O}(1)$ time using a shared table of size $\mathcal{O}(\sqrt{n})$. Descriptions of the sparse blocks are stored one after another as in the binary case.

This completes the description of the structure for a single $c \in [0, d-1]$. To access the relevant structure when answering a select query, we additionally store the pointers to the structures in an array of size $\mathcal{O}(d)$. Now we will argue that all structures can be constructed in $\mathcal{O}(\frac{N \log d}{\log n})$ time, but first let us recall what data is stored for a single $c \in [0, d-1]$:

- (1) an array storing the starting positions of all superblocks and pointers to their descriptions,
- (2) for every sparse superblock, a list of positions of every occurrence of c inside,
- (3) for every dense superblock, a bitvector with the i -th bit denoting if the i -th block inside is sparse,
- (4) an array storing the starting positions of all blocks relative to the starting position of their superblock,
- (5) for every sparse block, a list of positions of every occurrence of c inside relative to the starting position of its superblock.

First, we compute the starting positions of all superblocks by scanning the input while maintaining the number of occurrences of every $c \in [0, d-1]$ in its current superblock. These counters take $\mathcal{O}(d \log s) = o(\log n)$ bits together, and are packed in a single machine word. We read the input word-by-word and update the counters, which can be done in $\mathcal{O}(1)$ time after an appropriate preprocessing. Whenever we notice that a counter exceeds s , we output a new block. This requires just $\mathcal{O}(1)$ additional time per a superblock, assuming an appropriate preprocessing. The total time complexity is hence $\mathcal{O}(\frac{N \log d}{\log n})$ so far. Observe that we now know which superblock is sparse.

To compute the occurrences in sparse superblocks, we perform another word-by-word scan of the input. During the scan, we keep track of the current superblock for every $c \in [0, d-1]$, and if it is sparse, we extract and copy the starting positions of all occurrences of c . This requires just $\mathcal{O}(1)$ additional time per an occurrence of c in a sparse superblock, which sums up to $\mathcal{O}(\frac{N}{\log^2 n})$.

The remaining part is to split all dense superblocks into blocks. Again, we view the process as an automaton with one input tape, three output tapes corresponding to different parts of the structure, and a state consisting of $\mathcal{O}(\log \ell + s' \log \ell)$ bits. Here, we must be more careful than in the binary case: because we will be reading the whole input word-by-word, not just a single dense superblock, it might happen that ℓ is large. Therefore, before feeding the input into the automaton, we mask out all occurrences of c within sparse superblocks, which can be done in $\mathcal{O}(1)$ time per an input word after an appropriate preprocessing, if we keep track of the current superblock for every $c \in [0, d - 1]$ while scanning through the input. Then we can combine all d automata into a larger automaton equipped with $3d$ separate output tapes and a state consisting of $o(\log n)$ bits. We preprocess the behavior of the large automaton after reading a sequence of $\frac{1}{2} \log n$ bits given in the input tape for every initial state. The result is the new state, and the data written into each of the $3d$ output tapes. Now we again must be more careful: even though the output tapes are implemented as packed lists, and we need just $\mathcal{O}(1)$ time to append the preprocessed data to any them, we cannot afford to touch every output tape every time we read the next $\frac{1}{2} \log n$ bits from the input. Therefore, every output tape is equipped with an output buffer consisting of $\frac{1}{5} \frac{\log n}{d}$ bits, and all these output buffers are stored in a single machine word consisting of $\frac{1}{5} \log n$ bits. Then we modify the preprocessing: for every initial state, a sequence of $\frac{1}{5} \log n$ input bits, and the initial content of the output buffers, we compute the new state, the new content of the output buffers, and zero or more full chunks of $\frac{1}{5} \frac{\log n}{d}$ bits that should be written into the respective output tapes. Such preprocessing is still $\mathcal{O}(\sqrt{n})$, and now the additional time is just $\mathcal{O}(1)$ per each such chunk. Because we have chosen the parameters so that the size of the additional data in bits is $o(\frac{N}{d})$, the total time complexity is $\mathcal{O}(\frac{N \log d}{\log n})$ as claimed. \square