Fast Compressed Self-Indexes with Deterministic Linear-Time Construction*

J. Ian Munro[1], Gonzalo Navarro[2], and Yakov Nekrich[1]

1 **Cheriton School of Computer Science, University of Waterloo.** imunro@uwaterloo.ca, yakov.nekrich@googlemail.com.

2 **CeBiB — Center of Biotechnology and Bioengineering, Department of Computer Science, University of Chile.** gnavarro@dcc.uchile.cl.

Abstract

We introduce a compressed suffix array representation that, on a text T of length n over an alphabet of size σ, can be built in $O(n)$ deterministic time, within $O(n \log \sigma)$ bits of working space, and counts the number of occurrences of any pattern P in T in time $O(|P| + \log\log_w \sigma)$ on a RAM machine of $w = \Omega(\log n)$-bit words. This time is almost optimal for large alphabets ($\log \sigma = \Theta(\log n)$), and it outperforms all the other compressed indexes that can be built in linear deterministic time, as well as some others. The only faster indexes can be built in linear time only in expectation, or require $\Theta(n \log n)$ bits. For smaller alphabets, where $\log \sigma = o(\log n)$, we show how, by using space proportional to a compressed representation of the text, we can build in linear time an index that counts in time $O(|P|/\log_\sigma n + \log_\sigma^\epsilon n)$ for any constant $\epsilon > 0$. This is almost RAM-optimal in the typical case where $w = \Theta(\log n)$.

1998 ACM Subject Classification E.1 Data Structures; E.4 Coding and Information Theory

Keywords and phrases Succinct data structures; Self-indexes; Suffix arrays; Deterministic construction

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

The string indexing problem consists in preprocessing a string T so that, later, we can efficiently find occurrences of patterns P in T. The most popular solutions to this problem are suffix trees [39] and suffix arrays [26]. Both can be built in $O(n)$ deterministic time on a text T of length n over an alphabet of size σ, and the best variants can count the number of times a string P appears in T in time $O(|P|)$, and even in time $O(|P|/\log_\sigma n)$ in the word-RAM model if P is given packed into $|P|/\log_\sigma n$ words [34]. Once counted, each occurrence can be located in $O(1)$ time. Those optimal times, however, come with two important drawbacks:

- The variants with this counting time cannot be built in $O(n)$ worst-case time.
- The data structures use $\Theta(n \log n)$ bits of space.

The reason of the first drawback is that some form of perfect hashing is always used to ensure constant time per pattern symbol (or pack of symbols). The classical suffix trees and arrays with linear-time deterministic construction offer $O(|P| \log \sigma)$ or $O(|P| + \log n)$ counting time, respectively. More recently, those times have been reduced to $O(|P| + \log \sigma)$

* Funded with Basal Funds FB0001 and Fondecyt Grant 1-170048, Conicyt, Chile. A conference version of this paper appeared in *Proc. ISAAC 2017* [29].

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

[10] and even to $O(|P| + \log\log\sigma)$ [15]. Simultaneously with our work, a suffix tree variant was introduced by Bille et al. [7], which can be built in linear deterministic time and counts in time $O(|P|/\log_\sigma n + \log|P| + \log\log\sigma)$. All those indexes, however, still suffer from the second drawback, that is, they use $\Theta(n\log n)$ bits of space. This makes them impractical in most applications that handle large text collections.

Research on the second drawback dates back to almost two decades [33], and has led to indexes using $nH_k(T) + o(n(H_k(T)+1))$ bits, where $H_k(T) \le \log\sigma$ is the k-th order entropy of T [27], for any $k \le \alpha\log_\sigma n - 1$ and any constant $0 < \alpha < 1$. That is, the indexes use asymptotically the same space of the compressed text, and can reproduce the text and search it; thus they are called self-indexes. The fastest compressed self-indexes that can be built in linear deterministic time are able to count in time $O(|P|\log\log\sigma)$ [1] or $O(|P|(1+\log_w\sigma))$ [6]. There exist other compressed self-indexes that obtain times $O(|P|)$ [5] or $O(|P|/\log_\sigma n + \log_\sigma^\epsilon n)$ for any constant $\epsilon > 0$ [20], but both rely on perfect hashing and are not built in linear deterministic time. All those compressed self-indexes use $O(n\frac{\log n}{b})$ further bits to locate the position of each occurrence found in time $O(b)$, and to extract any substring S of T in time $O(|S| + b)$.

In this paper we introduce the first compressed self-index that can be built in $O(n)$ deterministic time (moreover, using $O(n\log\sigma)$ bits of space [31]) and with counting time $O(|P| + \log\log_w\sigma)$, where $w = \Omega(\log n)$ is the size in bits of the computer word. This is almost time-optimal on large alphabets, where $\log\sigma = \Theta(\log n)$, or when each pattern symbol is input in one computer word. More precisely, we prove the following result.

► **Theorem 1.** *On a RAM machine of $w = \Omega(\log n)$ bits, we can construct an index for a text T of length n over an alphabet of size $\sigma = o(n)$ in $O(n)$ deterministic time using $O(n\log\sigma)$ bits of working space. This index occupies $nH_k(T) + o(n\log\sigma) + O(n\frac{\log n}{b})$ bits of space for a parameter b and any $k \le \alpha\log_\sigma n - 1$, for any constant $0 < \alpha < 1$. The occurrences of a pattern string P can be counted in $O(|P| + \log\log_w\sigma)$ time, and then each such occurrence can be located in $O(b)$ time. An arbitrary substring S of T can be extracted in time $O(|S| + b)$.*

We obtain our results with a combination of the compressed suffix tree $\mathcal{T}$ of T and the Burrows-Wheeler transform $\overline{B}$ of the reversed text $\overline{T}$. We manage to simulate the suffix tree traversal for P, simultaneously on $\mathcal{T}$ and on $\overline{B}$. With a combination of storing deterministic dictionaries and precomputed rank values for sampled nodes of $\mathcal{T}$, and a constant-time method to compute an extension of partial rank queries that considers small ranges in $\overline{B}$, we manage to ensure that all the suffix tree steps, except one, require constant time. The remaining one is solved with general rank queries in time $O(\log\log_w\sigma)$. As a byproduct, we show that the compressed sequence representations that obtain those rank times [6] can also be built in linear deterministic time.

Compared with previous work, other indexes may be faster at counting, but either they are not built in linear deterministic time [5, 20, 34] or they are not compressed [34, 7]. Our index outperforms all the previous compressed [13, 1, 6], as well as some uncompressed [15], indexes that can be built deterministically.

Reusing some of our ideas, we also design an index for smaller alphabets, where $\log\sigma = o(\log n)$, if the symbols come packed in the computer words. Its search time is almost optimal in a RAM machine where $w = \Theta(\log n)$.

► **Theorem 2.** *On a RAM machine of $\Omega(\log n)$ bits, we can construct an index for a text T of length n over an alphabet of size σ, where $\log\sigma = o(\log n)$, in $O(n)$ deterministic time and working space. This index occupies $O(n(H_0(T)+1)) + o(n\log\sigma)$ bits of space. The*

	Compressed	Compact	Uncompressed
Deterministic	$\lvert P\rvert \log\log\sigma$ [1] $\lvert P\rvert(1+\log_w \sigma)$ [6] $\lvert P\rvert + \log\log_w \sigma$ **(ours)**	$\lvert P\rvert / \log n + \log^\epsilon n$ [20] (constant σ) $\lvert P\rvert / \log_\sigma n + \log_\sigma^\epsilon n$ **(ours)** ($\log\sigma = o(\log n)$)	$\lvert P\rvert + \log\log\sigma$ [15] $\lvert P\rvert / \log_\sigma n + \log\lvert P\rvert + \log\log\sigma$ [7]
Randomized	$\lvert P\rvert(1+\log\log_w \sigma)$ [6] $\lvert P\rvert$ [5]		$\lvert P\rvert / \log_\sigma n + \log_\sigma^\epsilon n$ [20, 34]

Table 1 Our results in context. The x axis refers to the space used by the indexes (compressed meaning $nH_k(T) + o(n\log\sigma)$ bits, compact meaning $O(n\log\sigma)$ bits, and uncompressed meaning $\Theta(n\log n)$ bits), and the y axis refers to the *linear-time* construction. In the cells we show the counting time for a pattern P. We only list the dominant alternatives, graying out those outperformed by our new results.

occurrences of a pattern string P can be counted in $O(|P|/\log_\sigma n + \log_\sigma^\epsilon n)$ time, for any constant $\epsilon > 0$, and then each such occurrence can be located in $O(\log_\sigma^\epsilon n)$ time. An arbitrary substring S of T can be extracted in time $O(|S|/\log_\sigma n)$.

Current indexes obtaining similar counting time require $O(n\log\sigma)$ construction time [20] or higher [34], or $O(n\log n)$ bits of space [34, 7].

2 Related Work

Let T be a string of length n over an alphabet of size σ that is indexed to support searches for patterns P. It is generally assumed that $\sigma = o(n)$, a reasonable convention we will follow. Searches typically require to *count* the number of times P appears in T, and then *locate* the positions of T where P occurs. The vast majority of the indexes for this task are suffix tree [39] or suffix array [26] variants.

The suffix tree can be built in linear deterministic time [39, 28, 38], even on arbitrarily large integer alphabets [11]. The suffix array can be easily derived from the suffix tree in linear time, but it can also be built independently in linear deterministic time [24, 23, 22]. In their basic forms, these structures allow counting the number of occurrences of a pattern P in T in time $O(|P|\log\sigma)$ (suffix tree) or $O(|P| + \log n)$ (suffix array). Once counted, the occurrences can be located in constant time each.

Cole et al. [10] introduced the *suffix trays*, a simple twist on suffix trees that reduces their counting time to $O(|P| + \log\sigma)$. Fischer and Gawrychowski [15] introduced the *wexponential search trees*, which yield suffix trees with counting time $O(|P| + \log\log\sigma)$ and support dynamism.

All these structures can be built in linear deterministic time, but require $\Theta(n\log n)$ bits of space, which challenges their practicality when handling large text collections.

Faster counting is possible if we resort to perfect hashing and give away the linear deterministic construction time. In the classical suffix tree, we can easily achieve $O(|P|)$ time by hashing the children of suffix tree nodes, and this is optimal in general. In the RAM model with word size $\Theta(\log n)$, and if the consecutive symbols of P come packed into $|P|/\log_\sigma n$ words, the optimal time is instead $O(|P|/\log_\sigma n)$. This optimal time was recently reached by

Navarro and Nekrich [34] (note that their time is not optimal if $w = \omega(\log n)$), with a simple application of weak-prefix search, already hinted in the original article [2]. However, even the randomized construction time of the weak-prefix search structure is $O(n \log^\epsilon n)$, for any constant $\epsilon > 0$. By replacing the weak-prefix search with the solution of Grossi and Vitter [20] for the last nodes of the search, and using a randomized construction of their perfect hash functions, the index of Navarro and Nekrich [34] can be built in linear randomized time and count in time $O(|P|/\log_\sigma n + \log_\sigma^\epsilon n)$. Only recently, simultaneously with our work, a deterministic linear-time construction algorithm was finally obtained for an index that counts in time $O(|P|/\log_\sigma n + \log |P| + \log\log\sigma)$ [7].

Still, these structures are not compressed. Compressed suffix trees and arrays appeared in the year 2000 [33]. To date, they take the space of the compressed text and replace it, in the sense that they can extract any desired substring of T; they are thus called self-indexes. The space occupied is measured in terms of the k-th order empirical entropy of T, $H_k(T) \le \log\sigma$ [27], which is a lower bound on the space reached by any statistical compressor that encodes each symbol considering only the k previous ones. Self-indexes may occupy as little as $nH_k(T) + o(n(H_k(T)+1))$ bits, for any $k \le \alpha \log_\sigma n - 1$, for any constant $0 < \alpha < 1$.

The fastest self-indexes with linear-time deterministic construction are those of Barbay et al. [1], which counts in time $O(|P| \log\log\sigma)$, and Belazzougui and Navarro [6, Thm. 7], which counts in time $O(|P|(1 + \log_w \sigma))$. The latter requires $O(n(1 + \log_w \sigma))$ construction time, but if $\log\sigma = O(\log w)$, its counting time is $O(|P|)$ and its construction time is $O(n)$.

If we admit randomized linear-time constructions, then Belazzougui and Navarro [6, Thm. 10] reach $O(|P|(1 + \log\log_w \sigma))$ counting time. At the expense of $O(n)$ further bits, in another work [5] they reach $O(|P|)$ counting time. Using $O(n \log\sigma)$ bits, and if P comes in packed form, Grossi and Vitter [20] can count in time $O(|P|/\log_\sigma n + \log_\sigma^\epsilon n)$, for any constant $\epsilon > 0$, however their construction requires $O(n \log\sigma)$ time.

Table 1 puts those results and our contribution in context. Our new self-index, with $O(|P| + \log\log_w \sigma)$ counting time, linear-time deterministic construction, and $nH_k(T) + o(n \log\sigma)$ bits of space, dominates all the compressed indexes with linear-time deterministic construction [1, 6], as well as some uncompressed ones [15] (to be fair, we do not cover the case $\log\sigma = O(\log w)$, as in this case the previous work [6, Thm. 7] already obtains our result). Our self-index also dominates a previous one with linear-time randomized construction [6, Thm. 10], which we incidentally show can also be built deterministically. The only aspect in which some of those dominated indexes may outperform ours is in that they may use $o(n(H_k(T)+1))$ [6, Thm. 10] or $o(n)$ [6, Thm. 7] bits of redundancy, instead of our $o(n \log\sigma)$ bits. For the case of small alphabets, where $\log\sigma = o(\log n)$, we also derive a compact index (using $O(n(H_0(T)+1)) + o(n \log\sigma)$ bits) that is built in linear deterministic time and counts in time $O(|P|/\log_\sigma n + \log_\sigma^\epsilon n)$. This is the same counting time obtained by Grossi and Vitter [20], who nevertheless require $O(n \log\sigma)$ construction time. In a sense, they have linear-time construction only for constant σ, whereas we extend it to the case $\log\sigma = o(\log n)$. Apart from generalizing their index to larger alphabets, our index outperforms the one we have sketched above by combining previous work [20, 34], since both take the same counting time but ours uses compact space.

3 Preliminaries

We denote by $T[i..]$ the suffix of $T[0, n-1]$ starting at position i and by $T[i..j]$ the substring that begins with $T[i]$ and ends with $T[j]$, $T[i..] = T[i]T[i+1]\ldots T[n-1]$ and $T[i..j] = T[i]T[i+1]\ldots T[j-1]T[j]$. We assume that the text T ends with a special symbol \$ that

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

lexicographically precedes all other symbols in T. The alphabet size is σ and symbols are integers in $[0..\sigma-1]$ (so \$ corresponds to 0). In this paper, as in the previous work on this topic, we use the word RAM model of computation. A machine word consists of $w = \Omega(\log n)$ bits and we can execute standard bit and arithmetic operations in constant time. We assume for simplicity that the alphabet size $\sigma = o(n)$ (otherwise the text is almost incompressible anyway [17]). Until Section 8, we also assume $\log \sigma = \omega(\log w)$, since otherwise our goal is already reached in previous work [6, Thm. 7].

3.1 Rank and Select Queries

We define three basic queries on sequences. Let $B[0..n-1]$ be a sequence of symbols over alphabet $[0..\sigma-1]$. The rank query, $\mathrm{rank}_a(i, B)$, counts how many times a occurs among the first $i+1$ symbols in B, $\mathrm{rank}_a(i, B) = |\{ j \le i,\ B[j] = a\}|$. The select query, $\mathrm{select}_a(i, B)$, finds the position in B where a occurs for the i-th time, $\mathrm{select}_a(i, B) = j$ iff $B[j] = a$ and $\mathrm{rank}_a(j, B) = i$. The third query is $\mathrm{access}(i, B)$, which returns simply $B[i]$.

We can answer access queries in $O(1)$ time and select queries in any $\omega(1)$ time, or vice versa, and rank queries in time $O(\log \log_w \sigma)$, which is optimal [6]. These structures use $n \log \sigma + o(n \log \sigma)$ bits, and we will use variants that require only compressed space. In this paper, we will show that those structures can be built in linear deterministic time.

An important special case of rank queries is the partial rank query, $\mathrm{rank}_{B[i]}(i, B)$, which asks how many times $B[i]$ occurrs in $B[0..i]$. Unlike general rank queries, partial rank queries can be answered in $O(1)$ time [6]. Such a structure can be built in $O(n)$ deterministic time and requires $O(n \log \log \sigma)$ bits of working and final space [31, Thm. A.4.1].

For this paper, we define a generalization of partial rank queries called interval rank queries, $\mathrm{rank}_a(i, j, B) = \langle \mathrm{rank}_a(i-1, B), \mathrm{rank}_a(j, B)\rangle$, from where in particular we can deduce the number of times a occurs in $B[i..j]$. If a does not occur in $B[i..j]$, however, this query just returns *null* (this is why it can be regarded as a generalized partial rank query).

In the special case where the alphabet size is small, $\log \sigma = O(\log w)$, we can represent B so that rank, select, and access queries are answered in $O(1)$ time [6, Thm. 7], but we are not focusing on this case in this paper, as the problem has already been solved for this case.

3.2 Suffix Array and Suffix Tree

The suffix tree [39] for a string $T[0..n-1]$ is a compacted digital tree on the suffixes of T, where the leaves point to the starting positions of the suffixes. We call X_u the string leading to suffix tree node u. The suffix array [26] is an array $SA[0..n-1]$ such that $SA[i] = j$ if and only if $T[j..]$ is the $(i+1)$-th lexicographically smallest suffix of T. All the occurrences of a substring P in T correspond to suffixes of T that start with P. These suffixes descend from a single suffix tree node, called the *locus* of P, and also occupy a contiguous interval in the suffix array SA. Note that the locus of P is the node u closest to the root for which P is a prefix of X_u. If P has no locus node, then it does not occur in T.

3.3 Compressed Suffix Array and Tree

A compressed suffix array (CSA) is a compact data structure that provides the same functionality as the suffix array. The main component of a CSA is the one that allows determining, given a pattern P, the suffix array range $SA[i..j]$ of the prefixes starting with P. Counting is then solved as $j-i+1$. For locating any cell $SA[k]$, and for extracting any substring S from T, most CSAs make use of a sampled array SAM_b, which contains the

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

values of $SA[i]$ such that $SA[i] \bmod b = 0$ or $SA[i] = n-1$. Here b is a tradeoff parameter: CSAs require $O(n\frac{\log n}{b})$ further bits and can locate in time proportional to b and extract S in time proportional to $b + |S|$. We refer to a survey [33] for a more detailed description.

A compressed suffix tree [37] is formed by a compressed suffix array and other components that add up to $O(n)$ bits. These include in particular a representation of the tree topology that supports constant-time computation of the preorder of a node, its number of children, its j-th child, its number of descendant leaves, and lowest common ancestors, among others [35]. Computing node preorders is useful to associate satellite information to the nodes.

Both the compressed suffix array and tree can be built in $O(n)$ deterministic time using $O(n \log \sigma)$ bits of space [31].

3.4 Burrows-Wheeler Transform and FM-index

The Burrows-Wheeler Transform (BWT) [8] of a string $T[0..n-1]$ is another string $B[0..n-1]$ obtained by sorting all possible rotations of T and writing the last symbol of every rotation (in sorted order). The BWT is related to the suffix array by the identity $B[i] = T[(SA[i]-1) \bmod n]$. Hence, we can build the BWT by sorting the suffixes and writing the symbols that precede the suffixes in lexicographical order.

The FM-index [12, 13] is a CSA that builds on the BWT. It consists of the following three main components:

- The BWT B of T.
- The array $Acc[0..\sigma-1]$ where $Acc[i]$ holds the total number of symbols $a < i$ in T (or equivalently, the total number of symbols $a < i$ in B).
- The sampled array SAM_b.

The interval of a pattern string $P[0..m-1]$ in the suffix array SA can be computed on the BWT B. The interval is computed backwards: for $i = m-1, m-2, \ldots$, we identify the interval of $P[i..m-1]$ in B. The interval is initially the whole $B[0..n-1]$. Suppose that we know the interval $B[i_1..j_1]$ that corresponds to $P[i+1..m-1]$. Then the interval $B[i_2..j_2]$ that corresponds to $P[i..m-1]$ is computed as $i_2 = Acc[a] + \text{rank}_c(i_1 - 1, B)$ and $j_2 = Acc[a] + \text{rank}_c(j_1, B) - 1$, where $a = P[i]$. Thus the interval of P is found by answering $2m$ rank queries. Any sequence representation offering rank and access queries can then be applied on B to obtain an FM-index.

An important procedure on the FM-index is the computation of the function LF, defined as follows: if $SA[j] = i+1$, then $SA[LF(j)] = i$. LF can be computed with access and partial rank queries on B, $LF(j) = \text{rank}_{B[j]}(i, B) + Acc[B[j]] - 1$, and thus constant-time computation of LF is possible. Using SAM_b and $O(b)$ applications of LF, we can locate any cell $SA[r]$. A similar procedure allows extracting any substring S of T with $O(b + |S|)$ applications of LF.

4 Small Interval Rank Queries

We start by showing how a compressed data structure that supports select queries can be extended to support a new kind of queries that we dub *small interval rank queries*. An interval query $\text{rank}_a(i, j, B)$ is a small interval rank query if $j - i \le \log^2 \sigma$. Our compressed index relies on the following result.

▶ **Lemma 3.** *Suppose that we are given a data structure that supports* access *queries on a sequence $C[0..m-1]$, on alphabet $[0..\sigma-1]$, in time t. Then, using $O(m \log\log \sigma)$ additional bits, we can support small interval rank queries on C in $O(t)$ time.*

Proof. We split C into groups G_i of $\log^2\sigma$ consecutive symbols, $G_i = C[i\log^2\sigma..(i+1)\log^2\sigma - 1]$. Let A_i denote the sequence of the distinct symbols that occur in G_i. Storing A_i directly would need $\log\sigma$ bits per symbol. Instead, we encode each element of A_i as its first position in G_i, which needs only $O(\log\log\sigma)$ bits. With this encoded sequence, since we have $O(t)$-time access to C, we have access to any element of A_i in time $O(t)$. In addition, we store a succinct SB-tree [19] on the elements of A_i. This structure uses $O(p\log\log u)$ bits to index p elements in $[1..u]$, and supports predecessor (and membership) queries in time $O(\log p/\log\log u)$ plus one access to A_i. Since $u = \sigma$ and $p \le \log^2\sigma$, the query time is $O(t)$ and the space usage is bounded by $O(m\log\log\sigma)$ bits.

For each $a \in A_i$ we also keep the increasing list $I_{a,i}$ of all the positions where a occurs in G_i. Positions are stored as differences with the left border of G_i: if $C[j] = a$, we store the difference $j - i\log^2\sigma$. Hence elements of $I_{a,i}$ can also be stored in $O(\log\log\sigma)$ bits per symbol, adding up to $O(m\log\log\sigma)$ bits. We also build an SB-tree on top of each $I_{a,i}$ to provide for predecessor searches.

Using the SB-trees on A_i and $I_{a,i}$, we can answer small interval rank queries $\mathrm{rank}_a(x, y, C)$. Consider a group $G_i = C[i\log^2\sigma..(i+1)\log^2\sigma - 1]$, an index k such that $i\log^2\sigma \le k \le (i+1)\log^2\sigma$, and a symbol a. We can find the largest $i\log^2\sigma \le r \le k$ such that $C[r] = a$, or determine it does not exist: First we look for the symbol a in A_i; if $a \in A_i$, we find the predecessor of $k - i\log^2\sigma$ in $I_{a,i}$.

Now consider an interval $C[x..y]$ of size at most $\log^2\sigma$. It intersects at most two groups, G_i and G_{i-1}. We find the rightmost occurrence of symbol a in $C[x..y]$ as follows. First we look for the rightmost occurrence $y' \le y$ of a in G_i; if a does not occur in $C[i\log^2\sigma..y]$, we look for the rightmost occurrence $y' \le i\log^2\sigma - 1$ of a in G_{i-1}. If this is $\ge x$, we find the leftmost occurrence x' of a in $C[x..y]$ using a symmetric procedure. When $x' \le y'$ are found, we can compute $\mathrm{rank}_a(x', C)$ and $\mathrm{rank}_a(y', C)$ in $O(1)$ time by answering partial rank queries (Section 3.1). These are supported in $O(1)$ time and $O(m\log\log\sigma)$ bits. The answer is then $\langle \mathrm{rank}_a(x', C) - 1, \mathrm{rank}_a(y', C)\rangle$, or *null* if a does not occur in $C[x..y]$. ◀

The construction of the small interval rank data structure is dominated by the time needed to build the succinct SB-trees [19]. These are simply B-trees with arity $O(\sqrt{\log u})$ and height $O(\log p/\log\log u)$, where in each node a Patricia tree for $O(\log\log u)$-bit chunks of the keys are stored. To build the structure in $O(\log p/\log\log u)$ time per key, we only need to build those Patricia trees in linear time. Given that the total number of bits of all the keys to insert in a Patricia tree is $O(\sqrt{\log u}\log\log u)$, we do not even need to build the Patricia tree. Instead, a universal precomputed table may answer any Patricia tree search for any possible set of keys and any possible pattern, in constant time. The size of the table is $O(2^{O(\sqrt{\log u}\log\log u)}\sqrt{\log u}) = o(u)$ bits (the authors [19] actually use a similar table to answer queries). For our values of p and u, the construction requires $O(mt)$ time and the universal table is of $o(\sigma)$ bits.

5 Compressed Index

We classify the nodes of the suffix tree $\mathcal{T}$ of T into heavy, light, and special, as in previous work [34, 31]. Let $d = \log\sigma$. A node u of $\mathcal{T}$ is *heavy* if it has at least d leaf descendants and *light* otherwise. We say that a heavy node u is *special* if it has at least two heavy children.

For every special node u, we construct a deterministic dictionary [21] D_u that contains the labels of all the heavy children of u: If the jth child of u, u_j, is heavy and the first symbol on the edge from to u to u_j is a_j, then we store the key a_j in D_u with j as satellite data. If a heavy node u has only one heavy child u_j and d or more light children, then we also store the data structure D_u (containing only that heavy child of u). If, instead, a heavy node has one heavy child and less than d light children, we just keep the index of the heavy child using $O(\log d) = O(\log\log\sigma)$ bits.

The second component of our index is the Burrows-Wheeler Transform $\overline{B}$ of the reverse text $\overline{T}$. We store a data structure that supports rank, partial rank, select, and access queries on $\overline{B}$. It is sufficient for us to support access and partial rank queries in $O(1)$ time and rank queries in $O(\log\log_w \sigma)$ time. We also construct the data structure described in Lemma 3, which supports small interval rank queries in $O(1)$ time. Finally, we explicitly store the answers to some rank queries. Let $\overline{B}[l_u..r_u]$ denote the range of $\overline{X_u}$, where $\overline{X_u}$ is the reverse of X_u, for a suffix tree node u. For all data structures D_u and for every symbol $a \in D_u$ we store the values of $\mathrm{rank}_a(l_u - 1, \overline{B})$ and $\mathrm{rank}_a(r_u, \overline{B})$.

Let us show how to store the selected precomputed answers to rank queries in $O(\log\sigma)$ bits per query. Following a known scheme [18], we divide the sequence $\overline{B}$ into chunks of size σ. For each symbol a, we encode the number d_k of times a occurs in each chunk k in a binary sequence $A_a = 01^{d_0}01^{d_1}01^{d_2}\ldots$. If a symbol $\overline{B}[i]$ belongs to chunk $k = \lfloor i/\sigma \rfloor$, then $\mathrm{rank}_a(i, \overline{B})$ is $\mathrm{select}_0(k+1, A_a) - k$ plus the number of times a occurs in $\overline{B}[k\sigma..i]$. The former value is computed in $O(1)$ time with a structure that uses $|A_a| + o(|A_a|)$ bits [9, 30], whereas the latter value is in $[0, \sigma]$ and thus can be stored in D_u using just $O(\log\sigma)$ bits. The total size of all the sequences A_a is $O(n)$ bits.

Therefore, D_u needs $O(\log\sigma)$ bits per element. The total number of elements in all the structures D_u is equal to the number of special nodes plus the number of heavy nodes with one heavy child and at least d light children. Hence all D_u contain $O(n/d)$ symbols and use $O((n/d)\log\sigma) = O(n)$ bits of space. Indexes of heavy children for nodes with only one heavy child and less than d light children add up to $O(n\log\log\sigma)$ bits. The structures for partial rank and small interval rank queries on $\overline{B}$ use $O(n\log\log\sigma)$ further bits. Since we assume that σ is $\omega(1)$, we can simplify $O(n\log\log\sigma) = o(n\log\sigma)$.

The sequence representation that supports access and rank queries on $\overline{B}$ can be made to use $nH_k(T) + o(n(H_k(T)+1))$ bits, by exploiting the fact that it is built on a BWT [6, Thm. 10].[1] We note that they use constant-time select queries on $\overline{B}$ instead of constant-time access, so they can use select queries to perform LF^{-1}-steps in constant time. Instead, with our partial rank queries, we can perform LF-steps in constant time (recall Section 3.4), and thus have constant-time access instead of constant-time select on $\overline{B}$ (we actually do not use query select at all). They avoid this solution because partial rank queries require $o(n\log\sigma)$ bits, which can be more than $o(n(H_k(T)+1))$, but we are already paying this price.

Apart from this space, array Acc needs $O(\sigma\log n) = O(n)$ bits and SAM_b uses $O(n\frac{\log n}{b})$. The total space usage of our self-index then adds up to $nH_k(T) + o(n\log\sigma) + O(n\frac{\log n}{b})$ bits.

6 Pattern Search

Given a query string P, we will find in time $O(|P| + \log\log_w \sigma)$ the range of the reversed string $\overline{P}$ in $\overline{B}$. A backward search for P in B will be replaced by an analogous backward search for $\overline{P}$ in $\overline{B}$, that is, we will find the range of $\overline{P[0..i]}$ if the range of $\overline{P[0..i-1]}$ is known. Let $[l_i..r_i]$ be the range of $\overline{P[0..i]}$. We can compute l_i and r_i from l_{i-1} and r_{i-1} as $l_i = Acc[a] + \mathrm{rank}_a(l_{i-1} - 1, \overline{B})$ and $r_i = Acc[a] + \mathrm{rank}_a(r_{i-1}, \overline{B}) - 1$, for $a = P[i]$. Using our auxiliary data structures on $\overline{B}$ and the additional information stored in the nodes of the suffix tree $\mathcal{T}$, we can answer the necessary rank queries in constant time (with one exception).

[1] In fact it is $nH_k(\overline{T})$, but this is $nH_k(T) + O(\log n)$ [12, Thm. A.3].

The idea is to traverse the suffix tree $\mathcal{T}$ in synchronization with the forward search on $\overline{B}$, until the locus of P is found or we determine that P does not occur in T.

Our procedure starts at the root node of $\mathcal{T}$, with $l_{-1} = 0$, $r_{-1} = n - 1$, and $i = 0$. We compute the ranges $\overline{B}[l_i..r_i]$ that correspond to $\overline{P[0..i]}$ for $i = 0, \ldots, |P| - 1$. Simultaneously, we move down in the suffix tree. Let u denote the last visited node of $\mathcal{T}$ and let $a = P[i]$. We denote by u_a the next node that we must visit in the suffix tree, i.e., u_a is the locus of $P[0..i]$. We can compute l_i and r_i in $O(1)$ time if $\text{rank}_a(r_{i-1}, \overline{B})$ and $\text{rank}_a(l_{i-1} - 1, \overline{B})$ are known. We will show below that these queries can be answered in constant time because either (a) the answers to rank queries are explicitly stored in D_u or (b) the rank query that must be answered is a small interval rank query. The only exception is the situation when we move from a heavy node to a light node in the suffix tree; in this case the rank query takes $O(\log\log_w \sigma)$ time. We note that, once we are in a light node, we need not descend in $\mathcal{T}$ anymore; it is sufficient to maintain the interval in $\overline{B}$.

For ease of description we distinguish between the following cases.

1. Node u is heavy and $a \in D_u$. In this case we identify the heavy child u_a of u that is labeled with a in constant time using the deterministic dictionary. We can also find l_i and r_i in time $O(1)$ because $\text{rank}_a(l_{i-1} - 1, \overline{B})$ and $\text{rank}_a(r_{i-1}, \overline{B})$ are stored in D_u.
2. Node u is heavy and $a \notin D_u$. In this case u_a, if it exists, is a light node. We then find it with two standard rank queries on $\overline{B}$, in order to compute l_i and r_i or determine that P does not occur in T.
3. Node u is heavy but we do not keep a dictionary D_u for the node u. In this case u has at most one heavy child and less than d light children. We have two subcases:
   a. If u_a is the (only) heavy node, we find this out with a single comparison, as the heavy node is identified in u. However, the values $\text{rank}_a(l_{i-1} - 1, \overline{B})$ and $\text{rank}_a(r_{i-1}, \overline{B})$ are not stored in u. To compute them, we exploit the fact that the number of non-a's in $\overline{B}[l_{i-1}..r_{i-1}]$ is less than d^2, as all the children apart from u_a are light and less than d. Therefore, the first and the last occurrences of a in $\overline{B}[l_{i-1}..r_{i-1}]$ must be at distance less than d^2 from the extremes l_{i-1} and r_{i-1}, respectively. Therefore, a small interval rank query, $\text{rank}_a(l_{i-1}, l_{i-1} + d^2, \overline{B})$, gives us $\text{rank}_a(l_{i-1} - 1, \overline{B})$, since there is for sure an a in the range. Analogously, $\text{rank}_a(r_{i-1} - d^2, r_{i-1}, \overline{B})$ gives us $\text{rank}_a(r_{i-1}, \overline{B})$.
   b. If u_a is a light node, we compute l_i and r_i with two standard rank queries on $\overline{B}$ (or we might determine that P does not appear in T).
4. Node u is light. In this case, $P[0..i-1]$ occurs at most d times in T. Hence $\overline{P[0..i-1]}$ also occurs at most d times in $\overline{T}$ and $r_{i-1} - l_{i-1} \leq d$. Therefore we can compute r_i and l_i in $O(1)$ time by answering a small interval rank query, $\langle \text{rank}_a(l_{i-1} - 1, \overline{B}), \text{rank}_a(r_{i-1}, \overline{B}) \rangle$. If this returns *null*, then P does not occur in T.
5. We are on an edge of the suffix tree between a node u and some child u_j of u. In this case all the occurrences of $P[0..i-1]$ in T are followed by the same symbol, c, and all the occurrences of $\overline{P[0..i-1]}$ are preceded by c in $\overline{T}$. Therefore $\overline{B}[l_{i-1}..r_{i-1}]$ contains only the symbol c. This situation can be verified with access and partial rank queries on $\overline{B}$: $\overline{B}[r_{i-1}] = \overline{B}[l_{i-1}] = c$ and $\text{rank}_c(r_{i-1}, \overline{B}) - \text{rank}_c(l_{i-1}, \overline{B}) = r_{i-1} - l_{i-1}$. In this case, if $a \neq c$, then P does not occur in T; otherwise we obtain the new range with the partial rank query $\text{rank}_c(r_{i-1}, \overline{B})$, and $\text{rank}_c(l_{i-1} - 1, \overline{B}) = \text{rank}_c(r_{i-1}, \overline{B}) - (r_{i-1} - l_{i-1} + 1)$. Note that if u is light we do not need to consider this case; we may directly apply case 4.

Except for the cases 2 and 3b, we can find l_i and r_i in $O(1)$ time. In cases 2 and 3b we need $O(\log\log_w \sigma)$ time to answer general rank queries. However, these cases only take place when the node u is heavy and its child u_a is light. Since all descendants of a light node are light, those cases occur only once along the traversal of P. Hence the total time to find the range of $\overline{P}$ in $\overline{B}$ is $O(|P| + \log\log_w \sigma)$. Once the range is known, we can count and report all occurrences of $\overline{P}$ in the standard way.

7 Linear-Time Construction

7.1 Sequences and Related Structures

Apart from constructing the BWT $\overline{B}$ of $\overline{T}$, which is a component of the final structure, the linear-time construction of the other components requires that we also build, as intermediate structures, the BWT B of T, and the compressed suffix trees $\overline{\mathcal{T}}$ and $\mathcal{T}$ of $\overline{T}$ and T, respectively. All these are built in $O(n)$ deterministic time and using $O(n \log \sigma)$ bits of space [31]. We also keep, on top of both $\overline{B}$ and B, $O(n \log\log \sigma)$-bit data structures able to report, for any interval $\overline{B}[i..j]$ or $B[i..j]$, all the distinct symbols from this interval, and their frequencies in the interval. The symbols are retrieved in arbitrary order. These auxiliary data structures can also be constructed in $O(n)$ time [31, Sec. A.5].

On top of the sequences B and $\overline{B}$, we build the representation that supports access in $O(1)$ and rank in $O(\log\log_w \sigma)$ time [6]. In their original paper, those structures are built using perfect hashing, but a deterministic construction is also possible [4, Lem. 11]; we give the details next.

The key part of the construction is that, within a chunk of σ symbols, we must build a virtual list I_a of the positions where each symbol a occurs, and provide predecessor search on those lists in $O(\log\log_w \sigma)$ time. We divide each list into blocks of $\log^2 \sigma$ elements, and create a succinct SB-tree [19] on the block elements, much as in Section 4. The search time inside a block is then $O(t)$, where t is the time to access an element in I_a, and the total extra space is $O(n \log\log \sigma)$ bits. If there is more than one block in I_a, then the block minima are inserted into a predecessor structure [6, App. A] that will find the closest preceding block minimum in time $O(\log\log_w \sigma)$ and use $O(n \log\log \sigma)$ bits. This structure uses perfect hash functions called $I(P)$, which provide constant-time membership queries. Instead, we replace them with deterministic dictionaries [21]. The only disadvantage of these dictionaries is that they require $O(\log \sigma)$ construction time per element, and since each element is inserted into $O(\log\log_w \sigma)$ structures $I(P)$, the total construction time per element is $O(\log \sigma \log\log_w \sigma)$. However, since we build these structures only on $O(n/\log^2 \sigma)$ block minima, the total construction time is only $O(n)$.

On the variant of the structure that provides constant-time access, the access to an element in I_a is provided via a permutation structure [32] which offers access time t with extra space $O((n/t) \log \sigma)$ bits. Therefore, for any $\log \sigma = \omega(\log w)$, we can have $t = O(\log\log_w \sigma)$ with $o(n \log \sigma)$ bits of space.

7.2 Structures D_u

The most complex part of the construction is to fill the data of the D_u structures. We visit all the nodes of $\mathcal{T}$ and identify those nodes u for which the data structure D_u must be constructed. This can be easily done in linear time, by using the constant-time computation of the number of descendant leaves. To determine if we must build D_u, we traverse its children $u_1, u_2, \ldots$ and count their descendant leaves to decide if they are heavy or light.

We use a bit vector D to mark the preorders of the nodes u for which D_u will be constructed: If p is the preorder of node u, then it stores a structure D_u iff $D[p] = 1$, in which case D_u is stored in an array at position $\mathrm{rank}_1(D, p)$. If, instead, u does not store D_u

but it has one heavy child, we store its child rank in another array indexed by $\mathrm{rank}_0(D,p)$, using $\log\log\sigma$ bits per cell.

The main difficulty is how to compute the symbols a to be stored in D_u, and the ranges $\overline{B}[l_u, r_u]$, for all the selected nodes u. It is not easy to do this through a preorder traversal of $\mathcal{T}$ because we would need to traverse edges that represent many symbols. Our approach, instead, is inspired by the navigation of the suffix-link tree using two BWTs given by Belazzougui et al. [3]. Let $\mathcal{T}_w$ denote the tree whose edges correspond to Weiner links between internal nodes in $\mathcal{T}$. That is, the root of $\mathcal{T}_w$ is the same root of $\mathcal{T}$ and, if we have internal nodes $u, v \in \mathcal{T}$ where $X_v = a \cdot X_u$ for some symbol a, then v descends from u by the symbol a in $\mathcal{T}_w$. We first show that the nodes of $\mathcal{T}_w$ are the internal nodes of $\mathcal{T}$. The inclusion is clear by definition in one direction; the other is well-known but we prove it for completeness.

▶ **Lemma 4.** *All internal nodes of the suffix tree $\mathcal{T}$ are nodes of $\mathcal{T}_w$.*

Proof. We proceed by induction on $|X_u|$, where the base case holds by definition. Now let a non-root internal node u of $\mathcal{T}$ be labeled by string $X_u = aX$. This means that there are at least two different symbols a_1 and a_2 such that both aXa_1 and aXa_2 occur in the text T. Then both Xa_1 and Xa_2 also occur in T. Hence there is an internal node u' with $X_{u'} = X$ in $\mathcal{T}$ and a Weiner link from u' to u. Since $|X_{u'}| = |X_u| - 1$, it holds by the inductive hypothesis that u' belongs to $\mathcal{T}_w$, and thus u belongs to $\mathcal{T}_w$ as a child of u'. ◀

We do not build $\mathcal{T}_w$ explicitly, but just traverse its nodes conceptually in depth-first order and compute the symbols to store in the structures D_u and the intervals in $\overline{B}$. Let u be the current node of $\mathcal{T}$ in this traversal and $\overline{u}$ its corresponding locus in $\overline{\mathcal{T}}$. Assume for now that $\overline{u}$ is a nod, too. Let $[l_u, r_u]$ be the interval of X_u in B and $[l_{\overline{u}}, r_{\overline{u}}]$ be the interval of the reverse string $X_{\overline{u}}$ in $\overline{B}$.[2] Our algorithm starts at the root nodes of $\mathcal{T}_w$, $\mathcal{T}$, and $\overline{\mathcal{T}}$, which correspond to the empty string, and the intervals in B and $\overline{B}$ are $[l_u, r_u] = [l_{\overline{u}}, r_{\overline{u}}] = [0, n-1]$. We will traverse only the heavy nodes, yet in some cases we will have to work on all the nodes. We ensure that on heavy nodes we work at most $O(\log\sigma)$ time, and at most $O(1)$ time on arbitrary nodes.

Upon arriving at each node u, we first compute its heavy children. From the topology of $\mathcal{T}$ we identify the interval $[l_i, r_i]$ for every child u_i of u, by counting leaves in the subtrees of the successive children of u. By reporting all the distinct symbols in $\overline{B}[l_{\overline{u}}..r_{\overline{u}}]$ with their frequencies, we identify the labels of those children. However, the labels are retrieved in arbitrary order and we cannot afford sorting them all. Yet, since the labels are associated with their frequencies in $\overline{B}[l_{\overline{u}}..r_{\overline{u}}]$, which match their number of leaves in the subtrees of u, we can discard the labels of the light children, that is, those appearing less than d times in $\overline{B}[l_{\overline{u}}..r_{\overline{u}}]$. The remaining, heavy, children are then sorted and associated with the successive heavy children u_i of u in $\mathcal{T}$.

If our preliminary pass marked that a D_u structure must be built, we construct at this moment the deterministic dictionary [21] with the labels a of the heavy children of u we have just identified, and associate them with the satellite data $\mathrm{rank}_a(l_{\overline{u}} - 1, \overline{B})$ and $\mathrm{rank}_a(r_{\overline{u}}, \overline{B})$. This construction takes $O(\log\sigma)$ time per element, but it includes only heavy nodes.

We now find all the Weiner links from u. For every (heavy or light) child u_i of u, we compute the list L_i of all the distinct symbols that occur in $B[l_i..r_i]$. We mark those symbols a in an array $V[0..\sigma-1]$ that holds three possible values: not seen, seen, and seen (at least)

[2] In the rest of the paper we wrote $\overline{B}[l_u..r_u]$ instead of $\overline{B}[l_{\overline{u}}..r_{\overline{u}}]$ for simplicity, but this may cause confusion in this section.

twice. If $V[a]$ is not seen, then we mark it as seen; if it is seen, we mark it as seen twice; otherwise we leave it as seen twice. We collect a list E_u of the symbols that are seen twice along this process, in arbitrary order. For every symbol a in E_u, there is an explicit Weiner link from u labeled by a: Let $X = X_u$; if a occurred in L_i and L_j then both aXa_i and aXa_j occur in T and there is a suffix tree node that corresponds to the string aX. The total time to build E_u amortizes to $O(n)$: for each child v of u, we pay $O(1)$ time for each child the node $\overline{v}$ has in $\overline{\mathcal{T}}$; each node in $\overline{\mathcal{T}}$ contributes once to the cost.

The targets of the Weiner links from u in $\mathcal{T}$ correspond to the children of the node $\overline{u}$ in $\overline{\mathcal{T}}$. To find them, we collect all the distinct symbols in $B[l_u..r_u]$ and their frequencies. Again, we discard the symbols with frequency less than d, as they will lead to light nodes, which we do not have to traverse. The others are sorted and associated with the successive heavy children of $\overline{u}$. By counting leaves in the successive children, we obtain the intervals $\overline{B}[l'_i..r'_i]$ corresponding to the heavy children $\overline{u}'_i$ of $\overline{u}$.

We are now ready to continue the traversal of $\mathcal{T}_w$: for each Weiner link from u by symbol a leading to a heavy node, which turns out to be the i-th child of $\overline{u}$, we know that its node in $\overline{\mathcal{T}}$ is $\overline{u}'_i$ (computed from $\overline{u}$ using the tree topology) and its interval is $\overline{B}[l'_i..r'_i]$. To compute the corresponding interval on B, we use the backward step operation, $B[x,y] = B[Acc[a] + \text{rank}_a(l_u - 1, B), Acc[a] + \text{rank}_a(r_u, B) - 1]$. This requires $O(\log\log_w \sigma)$ time, but applies only to heavy nodes. Finally, the corresponding node in $\mathcal{T}$ is obtained in constant time as the lowest common ancestor of the x-th and the y-th leaves of $\mathcal{T}$.

In the description above we assumed for simplicity that $\overline{u}$ is a node in $\overline{\mathcal{T}}$. In the general case $\overline{u}$ can be located on an edge of $\overline{\mathcal{T}}$. This situation arises when all occurrences of $\overline{X_u}$ in the reverse text $\overline{T}$ are followed by the same symbol a. In this case there is at most one Weiner link from u; the interval in $\overline{B}$ does not change as we follow that link.

A recursive traversal of $\mathcal{T}_w$ might require $O(n\sigma \log n)$ bits for the stack, because we store several integers associated to heavy children during the computation of each node u. We can limit the stack height by determining the largest subtree among the Weiner links of u, traversing all the others recursively, and then moving to that largest Weiner link target without recursion [3, Lem. 1]. Since only the largest subtree of a Weiner link target can contain more than half of the nodes of the subtree of u, the stack is guaranteed to be of height only $O(\log n)$. The space usage is thus $O(\sigma \log^2 n) = O(n \log \sigma)$.

As promised, we have spent at most $O(\log \sigma)$ time on heavy nodes, which are $O(n/d) = O(n/\log \sigma)$ in total, thus these costs add up to $O(n)$. All other costs that apply to arbitrary nodes are $O(1)$. The structures for partial rank queries (and the succinct SB-trees) can also be built in linear deterministic time, as shown in Section 4. Therefore our index can be constructed in $O(n)$ time.

8 A Compact Index for Small Alphabets

We now show that, if $\log \sigma = o(\log n)$, it is possible to obtain $O(|P|/\log_\sigma n + \log_\sigma^\epsilon n)$ search time with an index that uses $O(n \log \sigma)$ bits of space and is built in linear deterministic time. We start with a version that needs only $2nH_k(T) + o(n\log\sigma) + O(n)$ bits and $O(|P|/\log_\sigma n + \log^2 n)$ search time. A simple corollary yields $O(n \log \sigma)$ bits and $O(|P|/\log_\sigma n + \log n)$ search time. We then improve this result to reach $O(n(H_0(T)+1)) + o(n\log\sigma)$ bits and $O(|P|/\log_\sigma n + \log_\sigma^\epsilon n)$ search time, for any constant $\epsilon > 0$.

Structure

We store $\overline{B}$ in compressed form and a sample of the heavy nodes of $\mathcal{T}$. Following previous work [20, 34], we start from the root and store a deterministic dictionary [21] with all the highest suffix tree nodes v representing strings of depth $\geq \ell = \log_\sigma n$. The key associated with each node is a $(\log n)$-bit integer formed with the first ℓ symbols of the strings P_v. The satellite data are the length $|P_v|$, a position where P_v occurs in T, and the range $\overline{B}[l_v..r_v]$ of v. From each of those nodes v, we repeat the process with the first ℓ symbols that follow after P_v, and so on. The difference is that no light node will be inserted in those dictionaries. Let us charge the $O(\log n)$ bits of space to the children nodes, which are all heavy. If we count only the special nodes, which are $O(n/d)$, this amounts to $O((n \log n)/d)$ total bits and construction time. Recall that d is the maximum subtree size of light nodes. This time will use $d = \Theta(\log n)$ to have linear construction time and bit space, and thus will not take advantage of small rank interval queries.

There are, however, heavy nodes that are not special. These form possibly long chains between special nodes, and these will also induce chains of sampled nodes. While building the dictionaries for those nodes is trivial because they have only one sampled child, the total space may add up to $O(n \log \sigma)$ bits, if there are $\Theta(n)$ heavy nodes and the sampling chooses one out of ℓ in the chains. To avoid this, we increase the sampling step in those chains, enlarging it to $\ell' = \log n$. This makes the extra space spent in sampling heavy non-special nodes to be $O(n)$ bits as well.

In addition, we store the text T with a data structure that uses $nH_k(T) + o(n \log \sigma)$ for any $k = o(\log_\sigma n)$, and allows us extract $O(\log_\sigma n)$ consecutive symbols in constant time [14].

Queries

The search for P starts at the root, where its first ℓ symbols are used to find directly the right descendant node v in the dictionary stored at the root. If $|P_v| > \ell$, we directly compare the other $|P_v| - \ell$ symbols of P with the text, from the stored position where v appears, by chunks of ℓ symbols.[3] If the $|P_v|$ symbols coincide, we continue the search from v (ignore the chains for now); otherwise P does not occur in T.

When the next ℓ symbols of P are not found in the dictionary of the current node v, the search abandons the sampled nodes and enters a subtree of $O(d)$ light nodes. On this subtree, we continue using backward search on $\overline{B}$ from the interval $\overline{B}[l_v..r_v]$ stored at v, and do not use the suffix tree topology anymore. The number of backward steps to traverse such light nodes is $O(d)$, and can be performed in time $O(d \log \log_w \sigma)$. All the symbols between consecutive light nodes must be traversed in chunks of ℓ. Let v be our current node and u its desired child, and let $k+1$ be the number of symbols labeling the edge between them. After the first backward step from v to u leads us from the interval $\overline{B}[l_v..r_v]$ to $\overline{B}[l..r]$, we must perform k further backward steps, where each intermediate interval of $\overline{B}$ is formed by just one symbol. To traverse them fast, we first compute $t_l = \overline{SA}[l]$ and $t_r = \overline{SA}[r]$. The desired symbols are then $\overline{T}[t_l] = \overline{T}[t_r], \ldots, \overline{T}[t_l - k + 1] = \overline{T}[t_r - k + 1]$. We thus compare the suffixes $T[n-1-t_l..]$ and $T[n-1-t_r..]$ with what remains of P, by chunks of ℓ symbols, until finding the first difference; k is then the number of coincident symbols seen. If one of the two suffixes differs from P before the suffixes differ from each other, then P is not in T. Otherwise, we move (conceptually) to node u and consume the k coincident characters from

[3] A table of size $O(\sqrt{n})$ tells us the first symbol where any two chunks of $(\log_\sigma n)/2$ symbols differ. This is used to find the length of the shared prefix between the first chunks where P and T differ.

P. We can compute the new interval $[l_u..r_u] = [\overline{SA}^{-1}[t_l - k], \overline{SA}^{-1}[t_r - k]]$. Since $\overline{SA}$ and its inverse are computed in $O(b)$ time with arrays similar to SAM_b, we spend $O(b)$ time to cross each of the $O(d)$ edges in the final part of the search, plus $O(1)$ time per chunk of ℓ symbols in P.

When there are less than ℓ remaining symbols in P, we continue using backward search from the current interval $\overline{B}[l_v..r_v]$ (we might be at a sampled or at a light node). We just proceed symbolwise and complete the search in $O(\ell \log\log_w \sigma)$ time.

Let us now regard the case where we reach a sampled node v that starts a sampled chain. In this case the sampling step grows to ℓ'. We still compare P with a suffix of T where its heavy sampled child u appears, in chunks of ℓ symbols. If they coincide, we continue the search from u. Otherwise, we resume the search from $\overline{B}[l_v..r_v]$ using backward search. We might have to process ℓ' symbols of P, in time $O(\ell' \log\log_w \sigma)$, before reaching a light node. From the light node, we proceed as explained before.

Result

Overall, the total space is $2nH_k(T) + o(n \log \sigma) + O(n) + O((n \log n)/b)$ bits, and the construction time is $O(n)$. The search time is $O(|P|/\log_\sigma n + (\ell + \ell' + d) \log\log_w \sigma + db)$. We can, for example, choose $b = \Theta(\log n)$ to obtain the following result.

▶ **Theorem 5.** *On a RAM machine of $\Omega(\log n)$ bits, we can construct an index for a text T of length n over an alphabet of size $\sigma = o(n)$ in $O(n)$ deterministic time using $O(n \log \sigma)$ bits of working space. This index occupies $2nH_k(T) + o(n \log \sigma) + O(n)$ bits of space for any $k = o(\log_\sigma n)$. The occurrences of a pattern string P can be counted in $O(|P|/\log_\sigma n + \log^2 n)$ time, and then each such occurrence can be located in $O(\log n)$ time. An arbitrary substring S of T can be extracted in time $O(|S|/\log_\sigma n)$.*

Instead, by using $\ell' = \ell = O(\log_\sigma n)$ and $b = O(\log \sigma)$, we obtain the following result using compact space. Within this space we can handle larger σ values, as long as $\sigma \log n = O(n \log \sigma)$ (i.e., $\sigma = O(n)$).

▶ **Corollary 6.** *On a RAM machine of $\Omega(\log n)$ bits, we can construct an index for a text T of length n over an alphabet of size $\sigma = O(n)$ in $O(n)$ deterministic time using $O(n \log \sigma)$ bits of working space. The final index also occupies $O(n \log \sigma)$ bits. The occurrences of a pattern string P can be counted in $O(|P|/\log_\sigma n + \log n)$ time, and then each such occurrence can be located in $O(\log_\sigma n)$ time. An arbitrary substring S of T can be extracted in time $O(|S|/\log_\sigma n)$.*

8.1 Improving Space and Time

We now show how to further reduce the logarithmic extra penalty in Corollary 6, while using space proportional to a compressed representation of T. We will make use of compressed suffix arrays (CSAs) [20, 36] to represent SA and $\overline{SA}$, as well as the $O(n)$ further bits that, on top of SA, support a compressed suffix tree $\mathcal{T}$ [37] on T. We first explain how this structure works and how we build it in linear time and compact space; then we show how to use it to reduce the cost of the search algorithm at different points.

Compressed Suffix Arrays

We choose a CSA structure [20, 36] that uses $\frac{1}{\epsilon} nH_0(T) + o(n \log \sigma) + O(n)$ bits and computes any value of SA or SA^{-1} in time $O(\log_\sigma^\epsilon n)$, for any constant $0 < \epsilon \leq 1$. We describe the CSA of T (i.e., the one that represents SA); the one for $\overline{T}$ (representing $\overline{SA}$) is analogous.

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The main component of the CSA is a function Ψ, which is the inverse of LF: $\Psi(i) = SA^{-1}[SA[i] + 1 \bmod n]$. This function is represented across $\frac{1}{\epsilon}$ layers. In the first layer, the whole function Ψ is represented in an array $\Psi_0[0..n-1]$, with $\Psi_0[i] = \Psi(i)$. Further, a bitvector $V_0[0..n-1]$ indicates with $V_0[i] = 1$ the cells where $SA[i]$ is a multiple of $q = \log_\sigma^\epsilon n$. Those are the values of Ψ that will be represented in the next layer. The next layer stores the values $\Psi'(i) = \Psi^q(i)$, in a compacted array $\Psi_1[0..n/q-1]$, with $\Psi_1[\mathrm{rank}_1(V_0, i)] = \mathrm{rank}_1(V_0, \Psi'(i))$. This corresponds to the Ψ array of the text T_1 formed by chunks of q consecutive symbols of T. A bitvector $V_1[0..n/q-1]$ marks with $V_1[i]$ the positions where $SA_1[i]$ is a multiple of q, where SA_1 is the suffix array of T_1. After repeating the process a constant number of times, $L = \frac{1}{\epsilon} + 1$, the size of the arrays is $O(n/\log_\sigma^{1+\epsilon} n)$ and we can store the arrays SA_L and SA_L^{-1} explicitly using $O(n \log n / \log_\sigma^{1+\epsilon} n) = o(n \log \sigma)$ bits (recall that we assume $\log \sigma = o(\log n)$, thus $\log_\sigma n = \omega(1)$). Each of the arrays Ψ_l can be stored within $nH_0(T) + o(n \log \sigma) + O(n)$ bits by δ-encoding the consecutive differences $\Psi_l[i] - \Psi_l[i-1]$, while providing constant-time access to any cell of any Ψ_l [36]. Therefore, the total number of bits is $\frac{1}{\epsilon} nH_0(T) + o(n \log \sigma) + O(n)$ for any desired constant ϵ.

To find $SA[i]$ with this structure, we compute $i' = \Psi_0^k[i]$ until, for some $k < q$, it holds that $V_0[i'] = 1$. We then recursively compute $p_1 = SA_1[\mathrm{rank}_1(i', V_0)]$ from the next level, which corresponds to T_1. Then it holds that $SA[i] = p_1 \cdot q - k$. The total time incurred is $O(\frac{1}{\epsilon} \log_\sigma^\epsilon n)$. The computation of $SA^{-1}[i]$ is analogous, proceeding bottom-up [36].

We now show how to build this CSA in linear time and $O(n \log \sigma)$ bits of space. We first build B, the BWT of T, within those bounds. We also give constant-time support for select queries on B [6], which can be easily built in compact space and linear deterministic time. The function Ψ can be computed in constant time using select queries on B [25]: if $B[i] = c$, then $\Psi(i) = \mathrm{select}_c(i - Acc[c] + 1)$ (recall Section 3.4).

To build the first layer, we fill V_0 with zeros and traverse $\Psi^k(0)$ for $k = 1, 2, \ldots$, which virtually traverses the text forwards. For each k that is a multiple of q, we mark $V_0[\Psi^k(0)] = 1$ (i.e., $SA[\Psi^k(0)]$ is a multiple of q). Then, we traverse Ψ left to right, using δ-encoding on $\Psi(i) - \Psi(i-1)$, to produce the desired representation of the first-level array Ψ_0, together with the structures that support constant-time access on it [36]. All this process takes $O(n)$ time and compact space.

Now we must process the second level of the structure. To determine the 1s of the bitvector $V_1[0..n/q-1]$, we traverse the array Ψ_1, marking $V_1[i] = 1$ for those positions $i = \Psi_1^k(0)$ where k is a multiple of q. Once those are marked, we traverse Ψ_1 left to right in order to encode it using consecutive differences. Note that we do not yet have direct access to the values of Ψ_1, but we can recover any one in time $O(q)$ using $\Psi_1[i] = \mathrm{rank}_1(\Psi_0^q[\mathrm{select}_1(i, V_0)], V_0)$. Since Ψ_1 has only n/q cells, the total time to traverse it is also $O(n)$. Once we build the constant-time-access representation of Ψ_1, we use it to build Ψ_2, and so on.

The process is repeated a constant number of times, $\frac{1}{\epsilon}$. In total, we use only the $O(n \log \sigma)$ bits for B, the $O(n)$ bits for the bitvectors V_l, and the output size, and require $O(n)$ time. This structure computes in time $b = O(\log_\sigma^\epsilon n)$ any cell of SA, SA^{-1}, $\overline{SA}$, and $\overline{SA}^{-1}$. Next we show how to take advantage of this reduced value of b.

Speeding up the traversal of light nodes

We now reduce the $O(d \log \log_w \sigma + db) = O(db)$ time to traverse the light nodes to $O(\log d(\log \log_w \sigma + b \log d)) = O(b \log^2 d)$. For each light node v, we store the leaf in $\mathcal{T}$ where the heavy path starting at v ends. The heavy path chooses at each node the subtree with the most leaves, thus any traversal towards a leaf has to switch to another heavy path only $O(\log d)$ times. At each light node v, we go to the leaf u of its heavy path, obtain its position in T using the CSA, and compare the rest of P with the corresponding part of the suffix, by chunks of ℓ symbols. Once we determine the number k of symbols that coincide with P in the path from v to u, we perform a binary search for the highest ancestor v' of u where $|P_{v'}| - |P_v| \geq k$. If $|P_{v'}| - |P_v| > k$, then P does not appear in T (unless P ends at the k-th character compared, in which case the locus of P is v'). Otherwise, we continue the search from v'. Each binary search step requires $O(b)$ time to determine $|P_x|$ [37], thus the binary search takes time $O(b \log d)$. The single backward step performed from v' to move towards the new heavy path takes time $O(\log \log_w \sigma)$. Since we switch to another heavy path (the one starting at the child of v') $O(\log d)$ times, the total time is $O(b \log^2 d) = O(b(\log \log n)^2)$ as promised.

To store the leaf u corresponding to each light node v, we record the difference between the preorder numbers of u and v, which requires $O(\log d)$ bits. The node u is easily found in constant time from this information [37]. We have the problem, however, that we spend $O(\log d) = O(\log \log n)$ bits per light node, which adds up to $O(n \log \log n)$ bits. To reduce this to $O(n)$, we choose a second sampling step $e = O(\log \log n)$, and do not store this information on nodes with less than e leaves, which are called light-light. Those light nodes with e leaves or more are called light-heavy, and those with at least two light-heavy children are called light-special. There are $O(n/e)$ light-special nodes. We store heavy path information only for light-special nodes or for light-heavy nodes that are children of heavy nodes; both are $O(n/e)$ in total. A light-heavy node v that is not light-special has at most one light-heavy child u, and the heavy path that passes through v must continue towards u. Therefore, if it turns out that the search must continue from v after the binary search on the heavy path, then the search must continue towards the light-light children of v, therefore no heavy-path information is needed at node v because it will never be consulted.

Once we reach such a node v, its desired light-light node x is obtained with a backward step on $\overline{B}$, and from x we proceed as we did for Theorem 5 on light nodes, in total time $O(eb) = O(b \log \log n)$. Yet, we need the interval $\overline{B}[l_{\overline{v}}, r_{\overline{v}}]$ before we can start the search from v (here we call the extremes $l_{\overline{v}}$ and $r_{\overline{v}}$ to avoid confusion with the corresponding interval $B[l_v, r_v]$ in SA). The interval $SA[l_v, r_v]$ is indeed known by counting leaves in $\mathcal{T}$. To compute the corresponding range $\overline{SA}[l_{\overline{v}}, r_{\overline{v}}]$, we need to know the position p_v of the minimum in $\overline{SA}^{-1}[n-1-SA[i]]$ for $l_v \leq i \leq r_v$. We then have $l_{\overline{v}} = \overline{SA}^{-1}[n - SA[p_v] - |P_v|]$, and $r_{\overline{v}} = l_{\overline{v}} + r_v - l_v$.

A structure finding p_v is a *range minimum query (RMQ)* data structure on the array $C[i] = \overline{SA}^{-1}[n-1-SA[i]]$. We can easily build C in linear time by materializing SA and $\overline{SA}^{-1}$ in $O(n)$ time (by traversing Ψ and $\overline{\Psi}$). Then the RMQ structure is then built from C, in linear time and $O(n)$ bits of working space. After it is built, we can discard C and retain only $2n + o(n)$ bits for the final structure, which is sufficient to find any p_v in constant time given l_v and r_v [16]. Note, however, that we need $O(n \log n)$ bits of space for the construction of the intermediate array C.

Speeding up the traversal in a chain

Let us now reduce the $O(\ell' \log \log_w \sigma)$ time spent when leaving a chain to $O(b \log \ell')$. Given a sampled node v with exactly one heavy child, let u be the next sampled node, at distance ℓ' from v. Then, if the search does not continue from u, we can compute as before the number k of remaining symbols of P that coincide with P_u and then binary search the highest ancestor v' of u where $|P_{v'}| - |P_v| \geq k$. Thus we find in time $O(b \log \ell') = O(b \log \log n)$ the node in the path from where the search continues towards a light node. The way to compute the interval in $\overline{B}$ of that light node is the same we described above.

Recursing on the static dictionaries

Finally, let us reduce the $O(\ell \log\log_w \sigma)$ time to $O(\log_\sigma^\epsilon n)$. This cost is incurred when we have less than ℓ symbols left to examine in P and process them one by one. Instead, we will use a hierarchy of dictionaries that let us advance over progressively shorter pieces of P.

In addition to the dictionaries built to process chunks of $\ell_0 = \ell = \log_\sigma n$ symbols of P in constant time, we build similar dictionaries for $\ell_1 = \log_\sigma^{1-\epsilon} n$, $\ell_2 = \log_\sigma^{1-2\epsilon} n$, $\ldots$, $\ell_{1/\epsilon-1} = \log_\sigma^\epsilon n$. Since we only store special heavy nodes, there are only $O(n/d)$ nodes stored for each ℓ_k. The total space for all the dictionaries is then $O(\frac{1}{\epsilon} n) = O(n)$ bits.

Thus, when at a node v there are less than $\ell_0 = \ell$ pattern symbols left, and thus we cannot use the original deterministic dictionary, we try to use instead the one built for ℓ_1. After at most $\log_\sigma^\epsilon n$ chunks of length ℓ_1, the remaining length of P is $< \ell_1$. At this point we start using the dictionaries built for ℓ_2, and so on. We traverse at most $\log_\sigma^\epsilon n$ chunks of P of each length ℓ_k, and after $\frac{1}{\epsilon}$ lengths, we have only $\ell_{1/\epsilon-1} = \log_\sigma^\epsilon n$ symbols left in P. Only at this point we switch to the symbolwise traversal, if we have not yet reached a light node. Overall, the time $O(\ell \log\log_w \sigma)$ is reduced to $O(\frac{1}{\epsilon} \log_\sigma^\epsilon n)$.

Result

Our search time has been reduced to $O(|P|/\log_\sigma n + \frac{1}{\epsilon}\log_\sigma^\epsilon n + b(\log\log n)^2 + b\log\log n)$. For any desired constant $0 < \epsilon < 1$, this is $O(|P|/\log_\sigma n + \log_\sigma^\epsilon n)$ by using $b = \log_\sigma^{\epsilon'} n$ with any constant $0 < \epsilon' < \epsilon$. This yields our final result, Theorem 2. Note that, if $\log\sigma = \Theta(\log n)$, then the time is $O(|P| + (\log\log n)^2)$, already superseded by Theorem 1.

9 Conclusions

We have shown how to build, in $O(n)$ deterministic time and using $O(n\log\sigma)$ bits of working space, a compressed self-index for a text T of length n over an alphabet of size σ that searches for patterns P in time $O(|P| + \log\log_w \sigma)$, on a w-bit word RAM machine. This improves upon previous compressed self-indexes requiring $O(|P|\log\log\sigma)$ [1] or $O(|P|(1+\log_w \sigma))$ [6] time, on previous uncompressed indexes requiring $O(|P| + \log\log\sigma)$ time [15] (but that supports dynamism), and on previous compressed self-indexes requiring $O(|P|(1+\log\log_w \sigma))$ time and randomized construction (which we now showed how to build in linear deterministic time) [6]. The only indexes offering better search time require randomized construction [5, 20, 34] or $\Theta(n\log n)$ bits of space [34, 7].

For smaller alphabets where $\log\sigma = o(\log n)$, and if the symbols of P come in packed form, we also showed that, using space proportional to the zero-order empirical entropy of the text, we can build in $O(n)$ deterministic time an index that searches in time $O(|P|/\log_\sigma n + \log_\sigma^\epsilon n)$ for any constant $\epsilon > 0$. The only previous structures achieving this counting time either require $O(n\log\sigma)$ construction time or $O(n\log n)$ bits of space [20, 34].

It is not clear if $O(|P|)$ time, or even $O(|P|/\log_\sigma n)$, query time can be achieved with a linear deterministic construction time, even if we allow $O(n\log n)$ bits of space for the index (this was recently approached, but some additive polylog factors remain [7]). This is the most interesting open problem for future research.

References

1 J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.

2 D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. In *Proc. 18th Annual European Symposium on Algorithms (ESA)*, LNCS 6346, pages 427–438, 2010.
3 D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. Versatile succinct representations of the bidirectional Burrows-Wheeler transform. In *Proc. 21st Annual European Symposium on Algorithms (ESA)*, pages 133–144, 2013.
4 D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. Linear-time string indexing and analysis in small space. *CoRR*, abs/1609.06378, 2016.
5 D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):article 23, 2014.
6 D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):article 31, 2015.
7 P. Bille, I. L. Gørtz, and F. R. Skjoldjensen. Deterministic indexing for packed strings. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LIPIcs 78, page article 6, 2017.
8 M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
9 D. R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.
10 R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: Structures for faster text indexing. *Algorithmica*, 72(2):450–466, 2015.
11 M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.
12 P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
13 P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.
14 P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 371(1):115–121, 2007.
15 J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 9133, pages 160–171, 2015.
16 J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.
17 T. Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.
18 A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA)*, pages 368–373, 2006.
19 R. Grossi, A. Orlandi, R. Raman, and S. S. Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 517–528, 2009.
20 R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
21 T. Hagerup, P. Bro Miltersen, and R. Pagh. Deterministic dictionaries. *Journal of Algorithms*, 41(1):69 – 85, 2001.
22 J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.
23 D. K. Kim, J. S. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2-4):126–142, 2005.

24 P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

25 S. Lee and K. Park. Dynamic rank-select structures with applications to run-length encoded texts. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 95–106, 2007.

26 U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

27 G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

28 E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

29 I. Munro, G. Navarro, and Y. Nekrich. Fast compressed self-indexes with deterministic linear-time construction. In *Proc. 28th Annual International Symposium on Algorithms and Computation (ISAAC)*, LIPIcs 92, page article 57, 2017.

30 J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

31 J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 408–424, 2017.

32 J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

33 G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

34 G. Navarro and Y. Nekrich. Time-optimal top-k document retrieval. *SIAM Journal on Computing*, 46(1):89–113, 2017.

35 G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

36 K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

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

38 E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

39 P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory (FOCS)*, pages 1–11, 1973.

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany