Text Indexing and Searching in Sublinear Time

J. Ian Munro
Cheriton School of Computer Science, University of Waterloo
imunro@uwaterloo.ca

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

Yakov Nekrich
Department of Computer Science, Michigan Technological University
yakov.nekrich@googlemail.com

Abstract

We introduce the first index that can be built in $o(n)$ time for a text of length n, and can also be queried in $o(q)$ time for a pattern of length q. On an alphabet of size σ, our index uses $O(n \log \sigma)$ bits, is built in $O(n \log \sigma / \sqrt{\log n})$ deterministic time, and computes the number of occurrences of the pattern in time $O(q/\log_\sigma n + \log n \log_\sigma n)$. Each such occurrence can then be found in $O(\log n)$ time. Other trade-offs between the space usage and the cost of reporting occurrences are also possible.

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

Keywords and phrases data structures, string indexes

Digital Object Identifier 10.4230/LIPIcs...

Funding *Gonzalo Navarro*: Funded with Basal Funds FB0001, Conicyt, Chile

1 Introduction

We address the problem of indexing a text $T[0..n-1]$, over alphabet $[0..\sigma-1]$, in *sublinear* time on a RAM machine of $w = \Theta(\log n)$ bits. This is not possible when we build a classical index (e.g., a suffix tree [42] or a suffix array [26]) that requires $\Theta(n \log n)$ bits, since just writing the output takes time $\Theta(n)$. It is also impossible when $\log \sigma = \Theta(\log n)$ and thus just reading the $n \log \sigma$ bits of the input text takes time $\Theta(n)$. On smaller alphabets (which arise frequently in practice, for example on DNA, protein, and letter sequences), sublinear-time indexing becomes possible when the text comes packed in words of $\log_\sigma n$ characters and we build a *compressed* index that uses $o(n \log n)$ bits. For example, there exist various indexes that use $O(n \log \sigma)$ bits [35] (which is asymptotically the best worst-case size we can expect for an index on T) and could be built, in principle, in time $O(n/\log_\sigma n)$. Still, only linear-time indexing in compressed space had been achieved [3, 6, 30, 32] until the very recent result of Kempa and Kociumaka [24].

When the alphabet is small, one may also aim at RAM-optimal pattern search, that is, count the number of occurrences of a (packed) string $Q[0..q-1]$ in T in time $O(q/\log_\sigma n)$. There exist some classical indexes using $O(n \log n)$ bits and counting in time $O(q/\log_\sigma n + \mathrm{polylog}(n))$ [36, 11], as well as compressed ones [32].

In this paper we introduce the first index that can be *built and queried in sublinear time*. Our index, as explained, is compressed. It uses $O(n \log \sigma)$ bits and can be constructed in deterministic time $O(n \log \sigma/\sqrt{\log n})$. Thus the construction time is $O(n/\sqrt{\log n})$ when the alphabet size is a constant. Our index also supports counting queries in $o(q)$ time: it counts in optimal time plus an additive poly-logarithmic penalty, $O(q/\log_\sigma n + \log n \log_\sigma n)$. After counting the occurrences of Q, any such occurrence can be reported in $O(\log n)$ time.

A slightly larger and slower-to-build variant of our index uses $O(n(\sqrt{\log n \log \sigma} + \log \sigma \log^\varepsilon n))$ bits for any constant $0 < \varepsilon < 1/2$ and is built in time $O(n \log^{3/2} \sigma / \log^{1/2-\varepsilon} n)$. This index can report the occ pattern occurrences in time $O(q/\log_\sigma n + \sqrt{\log_\sigma n} \log\log n + \mathrm{occ})$.

As a comparison (see Table 1), the other indexes that count in time $O(q/\log_\sigma n + \mathrm{polylog}(n))$ use either more space ($O(n \log n)$ bits) and/or construction time ($O(n)$) [11, 36, 32]. The indexes using less space, on the other hand, use as little as $O(n \log \sigma)$ bits but are slower to build and/or to query [30, 29, 32, 3, 5, 6, 24]. A recent construction [24] is the only one able to build in sublinear time ($O(n \log \sigma/\sqrt{\log n})$) and to use compressed space ($O(n \log \sigma)$ bits), just like ours, but it is still unable to search in $o(q)$ time.

Those compressed indexes can then deliver each occurrence in $O(\log^\varepsilon n)$ time, or even in $O(1)$ time if a structure of $O(n \log^{1-\varepsilon} \sigma \log^\varepsilon n)$ further bits is added, though there is no sublinear-time construction for those extra structures either [38, 22].

Our technique is reminiscent to the Geometric BWT [15], where a text is sampled regularly, so that the sampled positions can be indexed with a suffix tree in sublinear space. In exchange, all the possible alignments of the pattern and the samples have to be checked in a two-dimensional range search data structure. To speed up the search, we use a data structure for LCE queries. An LCE data structure enables us to compute in constant time the longest common prefix of any two text positions. Using this information we can efficiently find the locus of each alignment from the previous one.

2 Preliminaries and LCE Queries

We denote by $|S|$ the number of symbols in a sequence S or the number of elements in a set S. For two strings X and Y, $LCP(X,Y)$ denotes the longest common prefix of X and Y. For a string X and a set of strings $\mathcal{S}$, $LCP(X,\mathcal{S}) = \max_{Y\in\mathcal{S}} LCP(X,Y)$, where we

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

Source	Construction time	Space (bits)	Query time (counting)
Classical [42, 27, 41, 19]	$O(n)$	$O(n \log n)$	$O(q \log \sigma)$
Cole et al. [17]	$O(n)$	$O(n \log n)$	$O(q + \log \sigma)$
Fischer & Gawrychowski [21]	$O(n)$	$O(n \log n)$	$O(q + \log\log \sigma)$
Bille et al. [11]	$O(n)$	$O(n \log n)$	$O(q/\log_\sigma n + \log q + \log\log \sigma)$
Classical + perfect hashing	$O(n)$ randomized	$O(n \log n)$	$O(q)$
Navarro & Nekrich [36]	$O(n)$ randomized	$O(n \log n)$	$O(q/\log_\sigma n + \log_\sigma^\varepsilon n)$
Barbay et al. [3]	$O(n)$	$O(n \log \sigma)$	$O(q \log\log \sigma)$
Belazzougui & Navarro [6]	$O(n)$	$O(n \log \sigma)$	$O(q(1 + \log_w \sigma))$
Munro et al. [30, 29]	$O(n)$	$O(n \log \sigma)$	$O(q + \log\log \sigma)$
Munro et al. [32]	$O(n)$	$O(n \log \sigma)$	$O(q + \log\log_w \sigma)$
Munro et al. [32]	$O(n)$	$O(n \log \sigma)$	$O(q/\log_\sigma n + \log_\sigma^\varepsilon n)$
Belazzougui & Navarro [6]	$O(n)$ randomized	$O(n \log \sigma)$	$O(q(1 + \log\log_w \sigma))$
Belazzougui & Navarro [5]	$O(n)$ randomized	$O(n \log \sigma)$	$O(q)$
Kempa and Kociumaka [24]	$O(n \log \sigma/\sqrt{\log n})$	$O(n \log \sigma)$	$O(q(1 + \log_w \sigma))$
Ours	$O(n \log \sigma/\sqrt{\log n})$	$O(n \log \sigma)$	$O(q/\log_\sigma n + \log n \cdot \log_\sigma n)$

Table 1 Previous and our results for index construction on a text of length n and a search pattern of length q, over an alphabet of size σ, on a RAM machine of w bits, for any constant $\varepsilon > 0$. Grayed rows are superseded by a more recent result in all aspects we consider. Note that $O(n)$-time randomized construction can be replaced by $O(n(\log\log n)^2)$ deterministic constructions [39].

compare lengths to take the maximum. We assume that the concepts associated with suffix trees [42] are known. The longest common extension (LCE) query on S asks for the length of the longest common prefix of suffixes $S[i..]$ and $S[j..]$, $LCE(i,j) = |LCP(S[i..], S[j..])|$. LCE queries were introduced by Landau and Vishkin [25]. Several recent publications demonstrate that LCE data structures can use $o(n)$ space and/or can be constructed in $o(n)$ time [40, 31, 24, 12]. The following result will play an important role in our construction.

▶ **Lemma 1.** *[24] Given a text T of length n over an alphabet of size σ, we can build an LCE data structure using $O(n \log \sigma)$ bits of space in $O(n/\log_\sigma n)$ time. This data structure supports LCE queries on T in $O(1)$ time.*

3 The General Approach

We divide the text $T[0..n-1]$, over alphabet $[0..\sigma-1]$, into *blocks* of $r = O(\log_\sigma n)$ consecutive symbols (to avoid tedious details, we assume that both r and $\log_\sigma n$ are integers and that n is divisible by both). The set $\mathcal{S}'$ consists of all the suffixes starting at positions ir, for $i = 0, 1, \ldots, n/r - 1$; these are called *selected* positions. Our data structure consists of the following three components.

1. The suffix tree $\mathcal{T}'$ for the suffixes starting at the selected positions, using $O((n/r)\log n)$ bits. Thus $\mathcal{T}'$ is a compacted trie for the suffixes in $\mathcal{S}'$. Suffixes are represented as strings of meta-symbols where every meta-symbol corresponds to a substring of $\log_\sigma n$ consecutive symbols. Deterministic dictionaries are used at the nodes to descend by the meta-symbols in constant time. Predecessor structures are also used at the nodes, to descend when less than a metasymbol of the pattern is left. Given a pattern Q, we can identify all selected suffixes starting with Q in $O(|Q|/\log_\sigma n)$ time, plus an $O(\log\log n)$ additive term coming from the predecessor operations at the deepest node.

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

2. A data structure on a set $\mathcal{Q}$ of points. Each point of $\mathcal{Q}$ corresponds to a pair (ind_i, rev_i) for $i = 1, \ldots, (n/r) - 1$ where ind_i is the index of the i-th selected suffix of T in the lexicographically sorted set $\mathcal{S}'$ and rev_i is an integer that corresponds to the reverse block preceding that i-th selected suffix in T. Our data structure supports two-dimensional range counting and reporting queries on $\mathcal{Q}$.
3. A data structure for *suffix jump* queries on $\mathcal{T}'$. Given a string $Q[0..q-1]$, its locus node u, and a positive integer $i \leq r-1$, a (suffix) i-jump query returns the locus node of $Q[i..q-1]$, or it says that $Q[i..q-1]$ does not prefix any string in $\mathcal{S}'$. The suffix jump structure has essentially the same functionality as the suffix links, but we do not store suffix links explicitly in order to save space and improve the construction time.

As described, $\mathcal{T}'$ is a compact trie over an alphabet of meta-symbols corresponding to strings of length $\log_\sigma n$. Therefore, whenever we speak of a *node* $u \in \mathcal{T}'$, we refer indistinctly to an explicit or an implicit node (i.e., in the middle of an edge, coming from compacting a unary path). Further, we cannot then properly speak of the "locus node" of a string Q, even if we identify meta-symbols with their forming strings, because $|Q|$ might not be a multiple of $\log_\sigma n$. Rather, the *locus of* Q will be denoted $u[l..s]$, where $u \in \mathcal{T}'$, called its *locus node*, is the deepest node whose string label is a prefix of Q and $[l..s]$ is the maximal interval such that the string labels of the children $u_l, \ldots, u_s$ of u are prefixed by Q.

Using our structure, we can find all the occurrences in T of a pattern $Q[0..q-1]$ whenever $q > r$. Occurrences of Q are classified according to their positions relative to selected symbols. An occurrence $T[f..f+q-1]$ of Q is an i-occurrence if $T[f+i]$ (corresponding to the i-th symbol of Q) is the leftmost selected symbol in $T[f..f+q-1]$.

First, we identify all 0-occurrences by looking for Q in $\mathcal{T}'$: We traverse the path corresponding to Q in $\mathcal{T}'$ to find $Q_0 = LCP(Q, \mathcal{S}')$, the longest prefix of Q that is in $\mathcal{T}'$, with locus $u_0[l_0..s_0]$. Let $q_0 = |Q_0|$; if $q_0 = q$, then $u_0[l_0..s_0]$ is the locus of Q and we count or report all its 0-occurrences as the positions of suffixes in the subtrees of $u_0[l_0..s_0]$.[1] If $q_0 < q$, there are no 0-occurrences of Q.

Next, we compute a 1-jump from u_0 to find the locus of $Q_0[1..] = Q[1..q_0-1]$ in $\mathcal{T}'$. If the locus does not exist, then there are no 1-occurrences of Q. If it exists, we traverse the path in $\mathcal{T}'$ for Q_1 starting from that locus, not redoing the path from the root. Let $Q_1 = Q[1..q_1-1] = LCP(Q[1..q-1], \mathcal{S}')$ be the longest prefix of $Q[1..q-1]$ found in $\mathcal{T}'$, with locus $u_1[l_1..s_1]$. If $q_1 < q$, then again there are no 1-occurrences of Q. If $q_1 = q$, then $u_1[l_1..s_1]$ is the locus of $Q[1..q-1]$. In this case, every 1-occurrence of Q corresponds to an occurrence of Q_1 in T that is preceded by $Q[0]$. We can identify them by answering a two-dimensional range query $[ind_1, ind_2] \times [rev_1, rev_2]$ where ind_1 (ind_2) is the leftmost (rightmost) leaf in the subtrees of $u_1[l_1..s_1]$ and rev_1 (rev_2) is the smallest (largest) integer value of any reverse block that starts with $Q[0]$.

We proceed and consider i-occurrences for $i = 2, \ldots, r-1$ using the same method. Suppose that we have already considered the possible j-occurrences of Q for $j = 0, \ldots, i-1$, so we have computed all the loci $u_j[l_j..s_j]$ of $Q_j = Q[j..q_j-1] = LCP(Q[j..q-1], \mathcal{S}')$. Further, let $q'_j \leq q_j$ be j plus the string depth of u_j, measured in symbols. This is the maximum number of symbols we can read from Q_j so that we reach a node of $\mathcal{T}'$. Let t be such that $q'_t = \max(q'_0, \ldots, q'_{i-1})$. We then compute the $(i-t)$-jump from u_t. If $Q[i..q'_t-1]$ is not found in $\mathcal{T}'$, then it is enough for us to know that $q_i < q'_t$ without actually finding the locus of Q_i. If $Q[i..q'_t-1]$ is found with locus node u, we traverse from u downwards to complete

[1] For fast counting, each node may also store the cumulative sum of its preceding siblings.

the path for $Q[i..q-1]$. We then find the locus $u_i[l_i..s_i]$ of $Q[i..q_i-1] = LCP(Q[i..q-1], \mathcal{S}')$. If $q_i = q$, then $Q[i..q-1]$ is found, so we count or report all i-occurrences by answering a two-dimensional query as described above.

Analysis. The total query time is $O(q/\log_\sigma n + r(\log\log n + t_q + t_s))$, where t_q and t_s are the times to answer a range query and to compute a suffix jump, respectively.

All the downward steps in the suffix tree amortize to $O(q/\log_\sigma n + r)$: we advance q'_t by $\log_\sigma n$ units in each downward step, but q'_t can be $(\log_\sigma n) - 1$ units less than the maximum position q_t we have reached up to now on Q (i.e., we take the suffix jump from u_t, whereas the actual locus with string depth q_t is $u_t[l_t..s_t]$). In addition we perform a predecessor step to find the ranges $[l_j..s_j]$ of the locus of each Q_j, which adds $O(r \log\log n)$ time. As said, the suffix tree (point 1) uses $O((n/r)\log n)$ bits.

The data structure of point 2 is a wavelet tree [14, 23, 34] built on $t = O(n/r)$ points. Its height is the logarithm of the y-coordinate range, $h = \log(\sigma^r) = O(r \log \sigma)$, and it uses $O(t \cdot h) = O(n \log \sigma) \subseteq O((n/r)\log n)$ bits. Such structure answers range counting queries in time $t_q = O(h) = O(r \log \sigma)$, thus $r \cdot t_q = O(r^2 \log \sigma)$, and reports each point in the range in time $O(h) = O(r \log \sigma)$.

In Sections 4 and 5 we show how to implement all the r suffix jumps (point 3) in time $r \cdot t_s = O(q/\log_\sigma n + r \log\log n)$, with a structure that uses $O((n/r)\log n)$ further bits.

Section 6 shows that the deterministic construction time of the structures of point 1 is $O(n(\log\log n)^2/r)$ and of point 3 is $O(n/r)$. The wavelet tree of point 2 can be built in time $O(t \cdot h/\sqrt{\log t}) = O(n \log \sigma/\sqrt{\log n})$ [33, 2].

Finally, since a pattern shorter than r may not cross a block boundary and thus we could miss occurrences, Section 7 describes a special index for small patterns. Its space and construction time is within those of point 3 for $r \le (1/4)\log_\sigma n$. This yields our first result.

▶ **Theorem 2.** *Let $0 < r < (1/4)\log_\sigma n$ be a parameter. Given a text T of length n over an alphabet of size σ, we can build an index using $O((n/r)\log n)$ bits in deterministic time $O(n((\log\log n)^2/r + \log\sigma/\sqrt{\log n}))$, so that it can count the number of occurrences of a pattern of length q in time $O(q/\log_\sigma n + r^2 \log\sigma + r\log\log n)$, and then report each such occurrence in time $O(r\log\sigma)$.*

If we set $r = \Theta(\log_\sigma n)$, we obtain a data structure with optimal asymptotic space usage.

▶ **Corollary 3.** *Given a text T of length n over an alphabet of size σ, we can build an index using $O(n\log\sigma)$ bits in deterministic time $O(n\log\sigma/\sqrt{\log n})$, so that it can count the number of occurrences of a pattern of length q in time $O(q/\log_\sigma n + \log n \log_\sigma n)$, and then report each such occurrence in time $O(\log n)$.*

We can improve the time of reporting occurrences by slightly increasing the construction time. Appendix A shows how to construct a range reporting data structure (point 2) that, after $t_q = O(\log\log n)$ time, can report each occurrence in constant time. The space of this structure is $O(n\log\sigma\log^\varepsilon n)$ bits and its construction time is $O((n \cdot r \cdot \log^2\sigma)/\log^{1-\varepsilon} n)$, for any constant $0 < \varepsilon < 1/2$. If we plug in this range reporting data structure into our index (i.e., replacing point 2 above), we obtain our second result.

▶ **Theorem 4.** *Let $0 < r < (1/4)\log_\sigma n$ be a parameter. Given a text T of length n over an alphabet of size σ, we can build an index using $O((n/r)\log n + n\log\sigma\log^\varepsilon n)$ bits in deterministic time $O(n((\log\log n)^2/r + (r\log^2\sigma)/\log^{1-\varepsilon} n))$, for any constant $0 < \varepsilon < 1/2$, so that it can count the occurrences of a pattern of length q in time $O(q/\log_\sigma n + r\log\log n)$, and then report each in $O(1)$ time.*

One interesting trade-off is when $r = \sqrt{\log_\sigma n}$. In this case the index uses $O(n(\sqrt{\log n}\log\sigma + \log\sigma\log^\varepsilon n))$ bits, can be constructed in $O((n\log^{3/2}\sigma)/\log^{1/2-\varepsilon} n)$ time, and reports the occ occurrences of a pattern of length q in time $O(q/\log_\sigma n + \sqrt{\log_\sigma n}\log\log n + \mathrm{occ})$.

4 Suffix Jumps

Now we show how suffix jumps can be implemented. The solution described in this section takes $O(\log n)$ time per jump $O((n/r)\log n)$ extra bits of space; it is used when $|Q| \geq \log^3 n$. This already provides us with an optimal solution because, in this case, the time of the r suffix jumps, $O(\log n \log_\sigma n)$, is subsumed by the time $O(q/\log_\sigma n)$ to traverse the pattern. In the next section we describe an appropriate method for short patterns.

Given a substring $Q_t[0..q_t - 1]$ of the original query Q, with known locus $u_t[l_t..s_t]$, we find the locus $v[l..s]$ of $Q_t[i..]$ or determine that it does not exist.

We compute the locus of $Q_t[i..]$ by applying Lemma 1 $O(\log n)$ times; note that we know the text position f_1 of an occurrence of Q_t because we know its locus $u_t[l_t..s_t]$ in $\mathcal{T}'$; therefore $Q_t[i..] = T[f_1 + i..]$. By binary search among the sampled suffixes (i.e., leaves of $\mathcal{T}'$), we identify in $O(\log n)$ time the suffix S_m that maximizes $|LCP(Q_t[i..], S_m)|$, because this measure decreases monotonically in both directions from S_m. At each step of the binary search we compute $\ell = |LCP(Q_t[i..], S)|$ for some suffix $S \in \mathcal{S}'$ using Lemma 1 and compare their $(\ell+1)$th symbols to decide the direction of the binary search. Once S_m is obtained we find, again with binary search, the smallest and largest suffixes $S_1, S_2 \in \mathcal{S}'$ such that $|LCP(S_1, S_m)| = |LCP(S_2, S_m)| = |LCP(Q_t[i..], S_m)|$; note $S_1 \leq S_m \leq S_2$.

Finally let v be the lowest common ancestor of the leaves that hold S_1 and S_2 in $\mathcal{T}'$. It then holds that $LCP(Q_t[i..], \mathcal{S}') = LCP(Q_t[i..], S_m)$, and v is its locus node. Further, the locus is $v[l..s]$, where S_1 and S_2 descend by the lth and sth children of v, respectively (we can find l and s in $O(1)$ time with level ancestor queries on $\mathcal{T}'$). If $|LCP(S_m, Q_t[i..])| = q_t - i = |Q_t[i..]|$, then $v[l..s]$ is also the locus of $Q_t[i..]$; otherwise $Q_t[i..]$ prefixes no string in $\mathcal{S}'$.

▶ **Lemma 5.** *Suppose that we know $Q_t[0..q_t - 1]$ and its locus in $\mathcal{T}'$. We can then compute $LCP(Q_t[i..q_t - 1], \mathcal{S}')$ and its locus in $\mathcal{T}'$ in $O(\log n)$ time, for any $0 \leq i \leq r - 1$.*

5 Suffix Jumps for Short Patterns

In this section we show how r suffix jumps can be computed in $O(|Q|/\log_\sigma n + r\log\log n)$ time when $|Q| \leq \log^3 n$. Our basic idea is to construct a set $\mathcal{X}_0$ of selected substrings with length up to $\log^3 n$. These are sampled at polylogarithmic-sized intervals from the sorted set $\mathcal{S}'$. We also create a superset $\mathcal{X} \supset \mathcal{X}_0$ that contains all the substrings that could be obtained by trimming the first $i \leq r - 1$ symbols from strings in $\mathcal{X}_0$. Using lexicographic naming and special dictionaries on $\mathcal{X}$, we pre-compute answers to all suffix jump queries for strings from $\mathcal{X}_0$. We start by reading the query string Q and trying to match Q, $Q[1..]$, $Q[2..]$ in $\mathcal{X}_0$. That is, for every $Q[i..q-1]$ we find $LCP(Q[i..q-1], \mathcal{X}_0)$ and its locus in $\mathcal{T}'$. With this information we can finish the computation of a suffix jump in $O(\log\log n)$ time, because the information on LCPs in $\mathcal{X}_0$ will narrow down the search in $\mathcal{T}'$ to a polylogarithmic sized interval, on which we can use the binary search of Section 4.

Data Structure. Let $\mathcal{S}''$ be the set obtained by sorting suffixes in $\mathcal{S}'$ and selecting every $(\log^{10} n)$th suffix. We denote by $\mathcal{X}$ the set of all substrings $T[i + f_1..i + f_2]$ such that the suffix $T[i..]$ is in the set $\mathcal{S}''$ and $0 \leq f_1 \leq f_2 \leq \log^3 n$. We denote by $\mathcal{X}_0$ the set of substrings $T[i..i+f]$ such that the suffix $T[i..]$ is in the set $\mathcal{S}''$ and $0 \leq f \leq \log^3 n$. Thus $\mathcal{X}_0$ contains all

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

prefixes of length up to $\log^3 n$ for all suffixes from $\mathcal{S}''$ and $\mathcal{X}$ contains all strings that could be obtained by suffix jumps from strings in $\mathcal{X}_0$.

We assign unique integer names to all substrings in $\mathcal{X}$: we sort $\mathcal{X}$ and then traverse the sorted list assigning a unique integer $\mathrm{num}(S)$ to each substring $S \in \mathcal{X}$. Our goal is to store pre-computed solutions to suffix jump queries. To this end, we keep three dictionaries:

- Dictionary D_0 contains the names $\mathrm{num}(S)$ for all $S \in \mathcal{X}_0$, as well as their loci in $\mathcal{T}'$.
- Dictionary D contains the names $\mathrm{num}(S)$ for all substrings $S \in \mathcal{X}$. For every entry $x \in D$, with $x = \mathrm{num}(S)$, we store (1) the length $\ell(S)$ of the string S, (2) the length $\ell(S')$ and the name $\mathrm{num}(S')$ where S' is the longest prefix of S satisfying $S' \in \mathcal{X}_0$, (3) for each j, $1 \le j \le r-1$, the name $\mathrm{num}(S[j..])$ of the string obtained by trimming the first j leading symbols of S if $S[j..]$ is in $\mathcal{X}$.
- Dictionary D_p contains $\mathrm{num}(S\alpha)$ for all pairs (x, α), where x is an integer and α is a string, such that the length of α is at most $\log_\sigma n$, $x = \mathrm{num}(S)$ for some $S \in \mathcal{X}$, and the concatenation $S\alpha$ is also in $\mathcal{X}$. D_p can be viewed as a (non-compressed) trie on $\mathcal{X}$.

Using D_p, we can navigate among the strings in $\mathcal{X}$: if we know $\mathrm{num}(S)$ for some $S \in \mathcal{X}$, we can look up the concatenation $S\alpha$ in $\mathcal{X}$ for any string α of length at most $\log_\sigma n$. The dictionary D enables us to compute suffix jumps between strings in $\mathcal{X}$: if we know $\mathrm{num}(S[0..])$ for some $S \in \mathcal{X}$, we can look up $\mathrm{num}(S[i..])$ in $O(1)$ time.

The set $\mathcal{S}''$ contains $O(\frac{n}{r \log^{10} n})$ suffixes. The set $\mathcal{X}$ contains $O(\log^6 n)$ substrings for every suffix in $\mathcal{S}''$. The space usage of dictionary D is $O(n/\log^4 n)$ words, dominated by item (3). The space of D_p is $O(n \log_\sigma n/(r \log^4 n))$ words, given by the number of strings in $\mathcal{X}$ times $\log_\sigma n$. This dominates the total space of our data structure, $O(n/\log^3 n)$ bits.

Suffix Jumps. Using the dictionary D, we can compute suffix jumps within $\mathcal{X}_0$.

► **Lemma 6.** *For any string Q with $r \le |Q| \le \log^3 n$, we can find the strings $P_i = LCP(Q[i..], \mathcal{X}_0)$, their lengths p_i and their loci in $\mathcal{T}'$, for all $1 \le i \le r-1$, in time $O(|Q|/\log_\sigma n + r \log\log_\sigma n)$.*

Proof. We find $P_0 = LCP(Q[0..q-1], \mathcal{X}_0)$ in $O(|P_0|/\log_\sigma n + \log\log_\sigma n)$ time: suppose that $Q[0..x]$ occurs in $\mathcal{X}_0$. We can check whether $Q[0..x + \log_\sigma n]$ also occurs in $\mathcal{X}_0$ using the dictionaries D_p and D_0. If this is the case, we increment x by $\log_\sigma n$. Otherwise we find with binary search, in $O(\log\log_\sigma n)$ time, the largest $f \le \log_\sigma n$ such that $Q[0..x+f]$ occurs in $\mathcal{X}_0$. Then $P_0 = Q[0..x+f] \in \mathcal{X}_0$, and its locus in $\mathcal{T}'$ is found in D_0.

When P_0, of length $p_0 = |P_0|$, and its name $\mathrm{num}(P_0)$ are known, we find $P_1 = LCP(Q[1..], \mathcal{X}_0)$: first we look up $v = \mathrm{num}(P_0[1..])$ in component (3) of D, then we look up in component (2) of D the longest prefix of the string with name v that is in $\mathcal{X}_0$. This is the 1-jump of P_0 in $\mathcal{X}_0$; now we descend as much as possible from there using D_p and D_0, as done to find P_0 from the root. We finally obtain $\mathrm{num}(P_1)$; its length p_1 and locus in $\mathcal{T}'$ are found in D (component (1)) and D_0, respectively.

We proceed in the same way as in Section 3 and find $LCP(Q[i..], \mathcal{X}_0)$ for $i = 2, \ldots, r-1$. The traversals in D_p amortize analogously to $O(|Q|/\log_\sigma n + r)$, and we have $O(r \log\log_\sigma n)$ further time to complete the r traversals. ◄

With all $LCP(Q[i..], \mathcal{X}_0)$ and their loci in $\mathcal{T}'$, we can compute suffix jumps in $\mathcal{S}'$.

► **Lemma 7.** *Suppose that we know $P_i = LCP(Q[i..q-1], \mathcal{X}_0)$ and its locus in $\mathcal{T}'$ for all $0 \le i \le r-1$. Assume we also know that $Q_t[0..q_t - 1] = Q[t..t + q_t - 1]$ prefixes a string in $\mathcal{S}'$ and its locus node $u_t \in \mathcal{T}'$. Then, given $j \le r-1$, we can compute $LCP(Q_t[j..], \mathcal{S}')$ and its locus in $\mathcal{T}'$, in $O(\log\log n)$ time.*

Proof. Let $v'[l'..s']$ be the locus of $LCP(Q_t[j..], \mathcal{X}_0) = LCP(Q[t+j..], \mathcal{X}_0)$ in $\mathcal{T}'$ and let $\ell = |LCP(Q_t[j..], \mathcal{X}_0)|$. If $\ell = q_t - j$, then $v'[l'..s']$ is the locus of $Q_t[j..]$ in $\mathcal{T}'$. Otherwise let v_+ denote the child of v' in $\mathcal{T}'$ that descends by $Q[t+j+\ell..t+j+\ell+\log_\sigma n - 1]$. If v_+ does not exist, then v' is the locus node v of $LCP(Q_t[j..], \mathcal{S}')$. We only have to find its children interval $[l..s]$ (which could expand $[l'..s']$) by a predecessor search on its children.

If v_+ exists, then the locus of $LCP(Q_t[j..], \mathcal{S}')$ is in the subtree $\mathcal{T}_{v_+}$ of $\mathcal{T}'$ rooted at v_+. By definition, $\mathcal{T}_{v_+}$ does not contain suffixes from $\mathcal{X}_0$. Hence $\mathcal{T}_{v_+}$ has $O(\log^{10} n)$ leaves. We then find $LCP(Q_t[j..], \mathcal{S}')$ among suffixes in $\mathcal{T}_{v_+}$ using the binary search method described in Section 4: we find S_1, S_m, and S_2 in time $O(\log\log^{10} n) = O(\log\log n)$. The locus $v[l..s]$ of $LCP(Q_t[j..], \mathcal{S}')$ is then the lowest common ancestor of the leaves that hold S_1 and S_2; l and s are the children S_1 and S_2 descend from. ◀

▶ **Lemma 8.** *Suppose that $|Q| \le \log^3 n$. Then we can find all the existing loci of $Q[i..]$ in $\mathcal{T}'$, for $0 \le i \le r-1$, in time $O(|Q|/\log_\sigma n + r\log\log n)$, using $O(n/\log^3 n)$ bits of space.*

6 Construction

Sampled suffix tree. We can view T as a string $\overline{T}$ of length n/r over an alphabet of size σ^r. Since $\overline{T}$ consists of $O(n/r)$ meta-symbols and each meta-symbol fits in a $\Theta(\log n)$-bit word, we can sort all meta-symbols in $O(n/r)$ time using RadixSort [18]. Thus we can generate $\overline{T}$ and construct its suffix tree $\mathcal{T}'$ in $O(n/r)$ time [19]. Further, we need $O((n/r)(\log\log n)^2)$ time to build the deterministic dictionaries and the predecessor data structures storing the children of each node [39, 4].

Suffix jumps. The lowest common ancestor and level ancestor structures [10, 8], which are needed in Section 4, are built in time $O(|\mathcal{T}'|) = O(n/r)$.

The sets of substrings and dictionaries D, D_0, and D_p described in Section 5 can be constructed as follows. Let $m = O(n/r)$ be the number of selected suffixes in $\mathcal{S}'$. The number of suffixes in $\mathcal{S}''$ is $O(m/\log^{10} n)$. The number of substrings associated with each suffix in $\mathcal{S}''$ is $O(\log^6 n)$ and their total length is $O(\log^9 n)$. The total number of strings in $\mathcal{X}_0$ is $O(m/\log^7 n)$ and their total length is $O(\frac{m}{\log^{10} n} \cdot \log^6 n) = O(m/\log^4 n)$. The number of strings in $\mathcal{X}$ is $k = O((m/\log^{10} n)\cdot\log^6 n) = O(m/\log^4 n)$ and their total length is $t = O((m/\log^{10} n)\cdot\log^9 n) = O(m/\log n)$. We can then collect all the strings $S \in \mathcal{X}$ from $T[i+f_1..i+f_2]$ for every sampled leaf of $\mathcal{T}'$ pointing to $T[i]$, sort them in $O(t) = o(m)$ time with RadixSort (the metasymbols fit in $O(\log n)$ bits [18]), remove repetitions, and finally assign them lexicographic names $\mathrm{num}(S)$. We keep a pointer to S in T for each $S \in \mathcal{X}$.

Next, we construct the dictionary D_0 that contains the names $\mathrm{num}(S)$ of those $S \in \mathcal{X}_0$. For every $x = \mathrm{num}(S)$ in D_0 we compute its locus $v[l..s]$ in $\mathcal{T}'$. The locus can be found in $O(|S|/\log_\sigma n + \log\log n)$ time by traversing $\mathcal{T}'$ from the root. This adds up to $O(|\mathcal{X}_0|\log^3 n) = o(m)$ time. Finally, D_0 is a deterministic dictionary on the keys $\mathrm{num}(S)$, so it can be constructed in $O(|\mathcal{X}_0|(\log\log n)^2) = o(m)$ deterministic time [39].

Similarly, D is a deterministic dictionary on k keys, which can be built in $O(k(\log\log n)^2) = o(m)$ time [39]. Since $\mathcal{X}$ is prefix-closed, we can use the pointers to the strings S and the dictionary D_0 to determine the longest prefix $S' \in \mathcal{X}_0$ of S by binary search on $\ell(S')$, in $O(k\log\log n)$ total time. When we generate strings of $\mathcal{X}$, we also record the information about suffix jumps (e.g., we store a pointer from each S to $S[1..]$ before sorting them, so later we can obtain $\mathrm{num}(S[1..])$ from S, then $\mathrm{num}(S[2..])$ from $S[1..]$, and so on). We can then easily traverse those suffixes to compute all relevant suffix jumps for each string $S \in \mathcal{X}$, in total time $O(kr) = o(m)$. We then have items (1)–(3) for all the elements of D.

Finally, we construct the dictionary D_p by inserting all strings in $\mathcal{X}$ into a trie data structure; at every node of this trie we store the name $\mathrm{num}(S)$ of the corresponding string S. Once $\mathcal{X}$ is sorted, the trie is easily built in $O(k)$ total time. Later, along a depth-first trie traversal we collect, for each node representing name y, its ancestors x up to distance $\log_\sigma n$ and the strings α separating x from y. All the pairs $(x, \alpha) \to y$ are then stored in D_p. Since $\mathcal{X}$ is prefix-closed, the trie contains $O(k)$ nodes, and we include $O(k \log_\sigma n)$ pairs in D_p. Since D_p is also a deterministic dictionary, it can be built in time $O(k \log_\sigma n (\log\log n)^2) = o(m)$.

The total time to build the data structures for suffix jumps is then $O(n/r + m) = O(n/r)$.

Range searches. As said, the wavelet tree can be built in time $O(n \log\sigma/\sqrt{\log n})$ [33, 2]. Appendix A shows that the time to build the data structure for faster reporting is $O(n \cdot r \cdot \log^2 \sigma / \log^{1-\varepsilon} n)$, for any constant $0 < \varepsilon < 1/2$.

7 Index for Small Patterns

The data structure for small query strings consists of two tables. Assume $r \le (1/4)\log_\sigma n$. We regard the text as an array $A[0..n/r]$ of length-$2r$ (overlapping) strings, $A[i] = T[ir..ir+2r-1]$. We build a table Tbl whose entries correspond to all strings of length $2r$: $\mathit{Tbl}[\alpha]$ lists all the positions i where $A[i] = \alpha$. Further, we build tables Tbl_j, for $1 \le j \le r$, containing all the possible length-j strings. Each entry $\mathit{Tbl}_j[\beta]$, with $|\beta| = j$, contains the list of length-$2r$ strings α such that $\mathit{Tbl}[\alpha]$ is not empty and β is a substring of α beginning within its first r positions (i.e., $\beta = \alpha[i..i+j-1]$ for some $0 \le i < r$).

Table Tbl has $\sigma^{2r} = O(\sqrt{n})$ entries, and overall contains n/r pointers to A, thus its total space is $O((n/r)\log n)$ bits. Tables Tbl_j add up to $O(\sigma^r) = O(n^{1/4})$ cells. Since each distinct string α of length $2r$ produces $O(r^2)$ distinct substrings, there can be only $O(\sigma^{2r} r^2) = O(\sqrt{n}\log^2_\sigma n)$ pointers in all the tables Tbl_j, for a total space of $o(n/r)$ bits.

To report the occurrences of $Q[0..q-1]$, we examine $\mathit{Tbl}_q[Q]$. For each string α in $\mathit{Tbl}_q[Q]$, we visit the entry $\mathit{Tbl}[\alpha]$ and report all the positions of $\mathit{Tbl}[\alpha]$ in A (with their offset).

To build Tbl, we can traverse A and add each i to the list of $\mathit{Tbl}[A[i]]$, all in $O(n/r)$ time. We then visit the slots of Tbl. For every α such that $\mathit{Tbl}[\alpha]$ is not empty, we consider all the sub-strings β of α starting within its first half and add α to $\mathit{Tbl}_{|\beta|}[\beta]$, recording also the corresponding offset of β in α (we may add the same α several times with different offsets). The time of this step is, as seen for the space, $O(\sigma^{2r} r^2) = O(\sqrt{n}\log^2_\sigma n) = o(n/r)$.

To support counting, $\mathit{Tbl}_q[Q]$ also stores the number of occurrences in T of each string Q.

▶ **Lemma 9.** *There exists a data structure that uses $O((n/r)\log n)$ bits and reports all* occ *occurrences of a query string Q in $O(\mathrm{occ})$ time if $|Q| \le r$, with $r \le (1/4)\log_\sigma n$. The data structure also computes* occ *in $O(1)$ time and can be built in time $O(n/r)$.*

8 Conclusion

We have described the first text index that can be built and queried in sublinear time. On a text of length n and alphabet of size σ, the index is built in $O(n\log\sigma/\sqrt{\log n})$ time, on a RAM machine of $\Theta(\log n)$ bits. This is sublinear for $\log\sigma = o(\sqrt{\log n})$. An index that is built in sublinear time must naturally use $o(n \log n)$ bits, hence our index is also compressed: our data structure has the asymptotically optimal space usage, $O(n\log\sigma)$ bits. Indeed, our index is the first one that simultaneously achieves three goals: sublinear construction time, asymptotically optimal space usage, and substring counting in nearly optimal time $O(q/\log_\sigma n + \log n \log_\sigma n)$ where q is the substring length. Previously described data structures with optimal (or even $O(n \log n)$) space usage either require $\Omega(n)$ construction time or $\Omega(q)$ time to count the occurrences of a substring.

We know no lower bound that prevents us from aiming at an index using the least possible space, $O(n \log \sigma)$ bits, the least possible construction time for this space in the RAM model, $O(n/\log_\sigma n)$, and the least possible counting time, $O(q/\log_\sigma n)$. Our index is the first one in breaking the $\Theta(n)$ construction time and $\Theta(q)$ query time barriers simultaneously, but it is open how close we can get to the optimal space and construction time.

A Range Reporting

In this section we prove a result on two-dimensional orthogonal range reporting queries. Our method builds upon previous work on wavelet tree construction [33, 2], applications of wavelet trees to range predecessor queries [7], and compact range reporting [14, 13].

▶ **Theorem 10.** *For a set of $t = O(n/r)$ points on a $t \times \sigma^{O(r)}$ grid, where $r \le (1/4)\log_\sigma n$, and for any constant $0 < \varepsilon < 1/2$, there is an $O(n \log \sigma \log^\varepsilon n)$-bit data structure that can be built in $O(n \cdot r \cdot \log^2 \sigma / \log^{1-\varepsilon} n)$ time and supports orthogonal range reporting queries in time $O(\log\log t + \mathrm{pocc})$ where pocc is the number of reported points.*

A.1 Base data structure

We are given a set $\mathcal{Q}$ of $t = O(n/r)$ points in $[0..t-1] \times [0..\sigma^{O(r)}]$. First we sort the points by x-coordinates (this is easily done by scanning the leaves of $\mathcal{T}'$, which are already sorted lexicographically by the selected suffixes), and keep the y-coordinates of every point in a sequence Y. Each element of Y can be regarded as a string of length $O(r)$ over an alphabet of size σ, or equivalently, an h-bit number where $h = O(r \log \sigma)$. Next we construct the range tree for Y using a method similar to the wavelet tree [23] construction algorithm. Let $Y(u_o) = Y$ for the root node u_o. We classify the elements of $Y(u_o)$ according to their highest bit and generate the corresponding subsequences of $Y(u_o)$, $Y(u_l)$ (highest bit zero) and $Y(u_r)$ (highest bit one), that must be stored in the left and right children of u, u_l and u_r, respectively. Then nodes u_l and u_r are recursively processed in the same manner. When we generate the sequence for a node u of depth d, we assign elements to $Y(u_l)$ and $Y(u_r)$ according to their d-th highest bit. We can exploit bit parallelism and pack $(\log n)/h$ y-coordinates into one word; therefore we can produce $Y(u_l)$ and $Y(u_r)$ from $Y(u)$ in $O(|Y(u)| \cdot h/\log n)$ time. The total time needed to generate all sequences $Y(u)$ is $O(t \cdot h \cdot (h/\log n)) = O((n \cdot r \cdot \log^2 \sigma)/\log n)$.

For every sequence $Y(u)$ we also construct an auxiliary data structure that supports three-sided queries. If u is a right child, we create a data structure that returns all elements in a range $[x_1, x_2] \times [0, h]$ stored in $Y(u)$. To this end, we divide $Y(u)$ into groups $G_i(u)$ of $g = (1/2)\log n$ consecutive elements (the last group may contain up to $2g$ elements). Let $\min_i(u)$ denote the smallest element in every group and let $Y'(u)$ denote the sequence of all $\min_i(u)$. We construct a data structure that supports three-sided queries on $Y'(u)$; it uses $O(|Y'(u)| \log n) = O((|Y(u)|/g) \log n) = O(|Y(u)|)$ bits and reports the k output points in $O(\log\log n + k)$ time; we can use any range minimum data structure for this purpose [9]. We can traverse $Y(u)$ and identify the smallest element in each group in $O(|Y(u)|h/\log n)$ time, by using small precomputed tables that process $(\log n)/2$ bits in constant time. This adds up to $O(t \cdot h^2/\log n) = O(n \cdot r \cdot \log^2 \sigma / \log n)$ time.

Since the number of points in $Y'(u)$ is $O(|Y(u)|/g)$, the data structure for $Y'(u)$ can be created in $O(|Y(u)|/g)$ time and uses $O((|Y(u)|/g) \log n) = O(|Y(u)|)$ bits, which adds up to $O((n \log \sigma)/\log n)$ construction time and $O(n \log \sigma)$ bits of space.

In order to save space, we do not store the y-coordinates of points in a group. The y-coordinate of each point in $G = G_i(u)$ is replaced with its rank, that is, with the number of points in G that have smaller y-coordinates. Each group G is divided into $(\log \sigma)/(2 \log\log n)$ subgroups, so that each subgroup contains $2r \log\log n$ consecutive points from G. We keep the rank of the smallest point from each subgroup of G in a sequence G^t. Since the ranks of points in a group are bounded by g and thus can be encoded with $\log g \le \log\log n$ bits, each subgroup can be encoded with less than $2r(\log\log n)^2$ bits. Hence we can store precomputed answers to all possible range minimum queries on all possible subgroups in a universal table of size $O(2^{2r(\log\log n)^2} \log^2 g) = o(n)$ bits. We can also store pre-computed answers for range minima queries on G^t using another small universal table: G^t is of length $(\log \sigma)/(2 \log\log n)$ and the rank of each minimum is at most g, so G^t can be encoded in at most $(\log \sigma)/2$ bits. This second universal table is then of size $O(2^{(\log \sigma)/2} \log^2 g) = o(n)$ bits.

A three-sided query $[x_1, x_2] \times [0, y]$ on a group G can then be answered as follows. We identify the point of smallest rank in $[x_1, x_2]$. This can be achieved with $O(1)$ table look-ups because a query on G can be reduced to one query on G^t plus a constant number of queries on sub-groups. Let x' denote the position of this smallest-rank point in $Y(u)$. We obtain the real y-coordinate of $Y(u)[x']$ using the translation method that will be described below. If the real y-coordinate of $Y(u)[x']$ does not exceed y, we report it and recursively answer three-sided queries $[x_1, x' - 1] \times [0, y]$ and $[x' + 1, x_2] \times [0, y]$. The procedure continues until all points in $[x_1, x_2] \times [0, y]$ are reported.

If u is a left child, we use the same method to construct the data structure that returns all elements in a range $[x_1, x_2] \times [y, +\infty)$ from $Y(u)$.

An orthogonal range reporting query $[x_1, x_2] \times [y_1, y_2]$ is then answered by finding the lowest common ancestor v of the leaves that hold y_1 and y_2. Then we visit the right child v_r of v, identify the range $[x'_1, x'_2]$ and report all points in $Y(v_r)[x'_1..x'_2]$ with y-coordinates that do not exceed y_2; here x'_1 is the index of the smallest x-coordinate in $Y(v_r)$ that is $\ge x_1$ and x'_2 is the index of the largest x-coordinate of $Y(v_r)$ that is $\le x_2$. We also visit the left child v_l of v, and answer the symmetric three-sided query. Finding x'_1 and x'_2 requires predecessor and successor queries on x-coordinates of any $Y(v_r)$; the needed data structures are described in Section A.3.

In total, the basic part of the data structure requires $O(n \log \sigma)$ bits of space and is built in time $O((n \cdot r \log^2 \sigma)/\log n)$.

A.2 Translating the answers

An answer to our three-sided query returns positions in $Y(v_l)$ (resp. in $Y(v_r)$). We need an additional data structure to translate such local positions into the points to be reported. While our wavelet tree can be used for this purpose, the cost of decoding every point would be $O(h)$. A faster decoding method [14, 37, 13] enables us to decode each point in $O(1)$ time. Below we describe how this decoding structure can be built within the desired time bounds.

Let us choose a constant $0 < \varepsilon < 1/2$ and, to simplify the description, assume that $\log_\sigma^\varepsilon n$ and $\log \sigma$ are integers. We will say that a node u is an x-node if the height of u is divisible by x. For an integer x the x-ancestor of a node v is the lowest ancestor w of v, such that w is an x-node. Let $d_k = h^{k\varepsilon}$ for $k = 0, 1, \ldots, \lceil 1/\varepsilon \rceil$. We construct sequences $UP(u)$ in all nodes u. $UP(u)$ enables us to move from a d_k-node to its d_{k+1}-ancestor: Let k be the largest integer such that u is a d_k-node and let v be the d_{k+1}-ancestor of u. We say that $Y(u)[i]$ corresponds to $Y(v)[j]$ if $Y(u)[i]$ and $Y(v)[j]$ represent the y-coordinates of the same point. Suppose that a three-sided query has returned position i in $Y(u)$. Using auxiliary structures, we find the corresponding position i_1 in the d_1-ancestor u_1 of u. Then we find i_2 that corresponds to i_1 in the d_2-ancestor u_2 of u_1. We continue in the same manner, at the k-th step moving from a d_k-node to its d_{k+1}-ancestor. After $O(1/\varepsilon)$ steps we reach the root node of the range tree.

It remains to describe the auxiliary data structures. To navigate from a node v to its ancestor u, v stores for every i in $Y(v)$ the corresponding position i' in $Y(u)$ (i.e., $Y(v)[i]$ and $Y(u)[i']$ are y-coordinates of the same point). In order to speed up the construction time, we store this information in two sequences. The sequence $Y(u)$ is divided into chunks; if u is a d_k-node, then the size of the chunk is $\Theta(2^{d_k})$. For every element in $Y(v)$ we store information about the chunk of its corresponding position in $Y(u)$ using the binary sequence $C(v)$: $C(v)$ contains a 1 for every element $Y(v)[i]$ and a 0 for every chunk in $Y(u)$ (0 indicates the end of a chunk). We store in $UP(v)[i]$ the relative value of its corresponding position in $Y(u)$. That is, if the element of $Y(u)$ that corresponds to $Y(v)[i]$ is in the jth chunk of $Y(u)$, then it is at $Y(u)[j \cdot 2^{d_k} + UP(v)[i]]$. In order to move from $Y(v)[i]$ in a node v to the corresponding position $Y(u)[i_k]$ in its d_k-ancestor u, we compute the target chunk in $Y(u)$, $j = \mathrm{select}_1(C(v), i) - i$, and set $i_k = j \cdot 2^{d_k} + UP(v)[i]$. Here select_1 finds the ith 1 in $C(v)$, and can be computed in constant time using $o(|C(v)|)$ bits on top of $C(v)$ [16, 28].

Since the tree contains h/d_{k-1} levels of t d_{k-1}-nodes, and the $UP(v)$ sequences of d_{k-1}-nodes v store numbers up to 2^{d_k}, the total space used by all $UP(v)$ sequences for all d_{k-1}-nodes v is $O(t \cdot (h/d_{k-1}) \cdot d_k) = O(t \cdot h^{1+\varepsilon})$ bits, because $d_k/d_{k-1} = h^{\varepsilon}$. For any such node v, with d_k-ancestor u, the total number of bits in $C(v)$ is $|Y(v)| + |Y(u)|/2^{d_k}$. There are at most 2^{d_k} nodes v with the same d_k-ancestor u. Hence, summing over all d_{k-1}-nodes v, all $C(v)$s use $t(h/d_{k-1}) + t(h/d_k) = O(t(h/d_{k-1}))$ bits. These structures are stored for all values $k-1 \in \{0, \ldots, \lceil 1/\varepsilon \rceil - 1\}$. Summing up, all sequences $C(v)$ use $O(t \cdot h)$ bits. The total space needed by auxiliary structures is then $O(t \cdot h^{1+\varepsilon}) = O(n \log^{1+\varepsilon/2} \sigma \log^{\varepsilon/2} n)$ bits, dominated by the sequences $UP(v)$. This can be written as $O(n \log \sigma \log^{\varepsilon} n)$ bits.

To produce the auxiliary structures, we need essentially that each d_k-node u distributes its positions in the corresponding $C(v)$ and $UP(v)$ structures in each of the next $h^{\varepsilon} - 1$ levels of d_{k-1}-nodes below u. Precisely, there are $2^{l \cdot d_{k-1}}$ d_{k-1}-nodes v at distance $l \cdot d_{k-1}$ from u, and we use $l \cdot d_{k-1}$ bits from the coordinates in $Y(u)[i]$ to choose the appropriate node v where $Y(u)[i]$ belongs. Doing this in sublinear time, however, requires some care.

Let us first consider the root u, the only d_k-node for $k = \lceil 1/\varepsilon \rceil$. We consider all the d_{k-1}-nodes v (thus, u is their only d_k-ancestor). These are nodes of height $l \cdot d_{k-1}$ for $l = 1, 2, \ldots, h^{\varepsilon} - 1$. In order to construct sequences $UP(v)$ in all nodes v on level $l \cdot d_{k-1}$ for a fixed l, we proceed as follows. The sequence $Y[u]$ is divided into chunks, so that each chunk contains 2^h consecutive elements. The elements $Y(u)[i]$ within each chunk are sorted with key pairs $(\mathrm{bits}((h^{\varepsilon} - l) \cdot d_{k-1}, Y(u)[i]), \mathrm{pos}(i, u))$ where $\mathrm{pos}(i, u) = i \bmod 2^h$ is the relative position of $Y(u)[i]$ in its chunk and $\mathrm{bits}(\ell, x)$ is the number that consists of the highest ℓ bits of x. We sort integer pairs in the chunk using a modification of the algorithm of Albers and Hagerup [1, Thm. 1] that runs in $O(2^h \frac{h^2}{\log n})$ time. Our modified algorithm works in the same way as the second phase of their algorithm, but we merge words in $O(1)$ time. Merging can be implemented using a universal look-up table that uses $O(\sqrt{n})$ words of space and can be initialized in $O(\sqrt{n} \log^3 n)$ time.

We then traverse the chunks and generate the sequences $UP(v)$ and $C(v)$ for all the nodes v on level $l \cdot d_{k-1}$. For each bit string of length $l \cdot d_{k-1}$, we say that v is the q-descendant of u if the path from u to v is labeled with q. The sorted list of pairs for each chunk of u is processed as follows. All the pairs $(q, \mathrm{pos}(i, u))$ (i.e., $q = \mathrm{bits}((h^{\varepsilon} - l)d_{k-1}, Y(u)[i])$) are consecutive after sorting, so we scan the list identifying the group for each value of q; let $n(q)$

be its number of pairs. Precisely, the points with value q must be stored at the q-descendant v of u (the consecutive values of q correspond, left-to-right, to the nodes v on level $l \cdot d_{k-1}$). For each group q, then, we identify the q-descendant v of u and append $n(q)$ 1-bits and one 0-bit to $C(v)$. We also append $n(q)$ entries to $UP(v)$ with the contents $\mathrm{pos}(i, u)$, in the same order as they appear in the chunk of u.

We need time $O(2^h \cdot h/\log n)$ to generate the pairs $(\mathrm{bits}(\cdot), \mathrm{pos}(\cdot))$ for the 2^h coordinates of each chunk, and to store the pairs in compact form, that is, $O(\log(n)/h)$ pairs per word. We can then sort the chunks in time $O(2^h \cdot h^2/\log n)$. We can generate the parts of sequences $C(v)$ and $UP(v)$ that correspond to a chunk for all nodes v on level $l \cdot d_{k-1}$ in $O(2^h + 2^h \cdot h/\log n) = O(2^h)$. Thus the total time needed to generate $UP(v)$ and $C(v)$ for all nodes v on level $l \cdot d_{k-1}$ and some fixed l is $O(t \log \sigma)$, where we remind that t is the total number of elements in the root node. The total time needed to construct $UP(v)$ and $C(v)$ for all d_{k-1}-nodes v is then $O(th^{2+\varepsilon}/\log n)$.

Now let u be an arbitrary d_k-node. Using almost the same method as above, we can produce sequences $UP(v)$ and $C(v)$ for all (d_{k-1})-nodes v, such that u is a d_k-ancestor of v. There are only two differences with the method above. First, we divide the sequence $Y(u)$ into chunks of size 2^{d_k}. Second, the sorting of elements in a chunk is not based on the highest bits, but on a less significant chunk of bits: the pairs are now $(\mathrm{bitval}(Y(u)[i]), \mathrm{pos}(i, u))$. If the bit representation of $Y(u)[i]$ is $b_1 b_2 \ldots b_d$, then $\mathrm{bitval}(Y(u)[i])$ is the integer with bit representation $b_{f+1} b_{f+2} \ldots b_{f+d_k}$ where f is the depth of the node u in the range tree. The total time needed to produce $C(v)$ and $UP(v)$ is $O(|Y(u)|d_k/\log n + |Y(u)|d_k^2/\log n)$, the first term to create the pairs and the second to sort the chunks and produce $C(v)$ and $UP(v)$. The number of different elements in all d_k-nodes is $O(t \cdot h/d_k)$, and each produces the sequences of h^ε levels of d_{k-1}-nodes. Hence the time needed to produce the sequences for all d_{k-1}-nodes is $O((t \cdot h)/d_k \cdot h^\varepsilon \cdot d_k^2/\log n) = O(t \cdot h^{1+\varepsilon} \cdot d_k/\log n) = O(t(h^2/\log n)h^\varepsilon)$. The complexity stays the same after adding up the $1/\varepsilon$ values of k: $O(t \cdot h^{2+\varepsilon}/\log n) = O((n/r)r^2 \log^2 \sigma \log^\varepsilon n/\log n) = O((n \cdot r \cdot \log^2 \sigma/\log^{1-\varepsilon} n)$.

The data structure supporting select queries on $C(v)$ can be built in $O(|C(v)|/\log n)$ time [33, Thm. 5]. This amounts to $O(th/\log n) = O(n/\log_\sigma n)$ further time.

A.3 Predecessors and successors of x-coordinates

Now we describe how predecessor and successor queries on x-coordinates of points in $Y(u)$ can be answered for any node u in time $O(\log \log n)$.

We divide the sequence $Y(u)$ into blocks, so that each block contains $\log n$ points. We keep the minimum x-coordinate from every block in a predecessor data structure $Y^b(u)$. In order to find the predecessor of x in $Y(u)$, we first find its predecessor x'' in $Y^b(u)$; then we search the block of x'' for the predecessor of x in $Y(u)$.

The predecessor data structure finds x'' in $O(\log \log n)$ time. We compute the x-coordinate of any point in $Y(u)$ in $O(1)$ time as shown above. Hence the predecessor of x in a block is found in $O(\log \log n)$ time too, using binary search. We find the successor analogously.

The sampled predecessor/successor data structures store $O((n/r)(r \log \sigma)/\log n) = O(n/\log_\sigma n)$ elements over all the levels. An appropriate construction [20, Thm. 4.1] builds them in linear time ($O(n/\log_\sigma n)$) and space ($O(n \log \sigma)$ bits), once they are sorted.

References

1 S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Information and Computation*, 136(1):25–51, 1997.

2 M. Babenko, P. Gawrychowski, T. Kociumaka, and T. Starikovskaya. Wavelet trees meet suffix trees. In *Proc. 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 572–591, 2015.
3 J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.
4 D. Belazzougui, P. Boldi, and S. Vigna. Dynamic z-fast tries. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 159–172, 2010.
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 D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2053–2071, 2016.
8 M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.
9 M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th Latin American Symposiumon Theoretical Informatics (LATIN)*, pages 88–94, 2000.
10 M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.
11 P. Bille, I. L. Gørtz, and F. R. Skjoldjensen. Deterministic indexing for packed strings. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 6:1–6:11, 2017.
12 O. Birenzwige, S. Golan, and E. Porat. Locally consistent parsing for text indexing in small space. In *Proc. 31st ACM-SIAM Symposium on Discrete Algorithms, (SODA)*, pages 607–626, 2020.
13 T. M. Chan, K. G. Larsen, and M. Patrascu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.
14 B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.
15 Y.-F. Chien, W.-K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. Geometric BWT: Compressed text indexing via sparse suffixes and range searching. *Algorithmica*, 71(2):258–278, 2015.
16 D. R. Clark. *Compact PAT Trees.* PhD thesis, University of Waterloo, Canada, 1996.
17 R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: Structures for faster text indexing. *Algorithmica*, 72(2):450–466, 2015.
18 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 3rd edition, 2009.
19 M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.
20 G. Feigenblat, E. Porat, and A. Shiftan. Linear time succinct indexable dictionary construction with applications. In *Proc. 26th Data Compression Conference (DCC)*, pages 13–22, 2016.
21 J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 160–171, 2015.
22 R. González, G. Navarro, and H. Ferrada. Locally compressed suffix arrays. *ACM Journal of Experimental Algorithmics*, 19(1):article 1, 2014.
23 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
24 D. Kempa and T. Kociumaka. String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure. In *Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 756–767, 2019.

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

25 G. M. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988.

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

28 J. I. Munro. Tables. In *Proc. 16th FSTTCS*, pages 37–42, 1996.

29 J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. *CoRR*, abs/1607.04346, 2016.

30 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.

31 J. I. Munro, G. Navarro, and Y. Nekrich. Text indexing and searching in sublinear time. *CoRR*, abs/1712.07431, 2017.

32 J. I. Munro, G. Navarro, and Y. Nekrich. Fast compressed self-indexes with deterministic linear-time construction. *Algorithmica*, 82(2):316–337, 2020.

33 J. I. Munro, Y. Nekrich, and J. S. Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016.

34 G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

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

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

37 Y. Nekrich. Orthogonal range searching in linear and almost-linear space. *Computational Geometry*, 42(4):342–351, 2009.

38 S. S. Rao. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters*, 82(6):307–311, 2002.

39 M. Ruzic. Constructing efficient dictionaries in close to sorting time. In *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP A)*, pages 84–95 (part I), 2008.

40 Y. Tanimura, T. Nishimoto, H. Bannai, S. Inenaga, and M. Takeda. Small-space LCE data structure with constant-time queries. In *Proc. 42nd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 10:1–10:15, 2017.

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

42 P. Weiner. Linear pattern matching algorithms. In *Proc. 14th IEEE Symposium on Foundations on Computer Science (FOCS)*, pages 1–11, 1973.

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