The VLDB Journal (2025) 34:58
https://doi.org/10.1007/s00778-025-00935-7

REGULAR PAPER

Text indexing for long patterns using locally consistent anchors

Lorraine A. K. Ayad[1] · Grigorios Loukides[2] · Solon P. Pissis[3]

Received: 16 July 2024 / Revised: 21 June 2025 / Accepted: 1 July 2025 / Published online: 15 July 2025

Abstract
In many real-world database systems, a large fraction of the data is represented by strings: sequences of letters over some alphabet. This is because strings can easily encode data arising from different sources. It is often crucial to represent such string datasets in a compact form but also to *simultaneously* enable fast pattern matching queries. This is the classic text indexing problem. The four absolute measures anyone should pay attention to when designing or implementing a text index are: **(i)** index space; **(ii)** query time; **(iii)** construction space; and **(iv)** construction time. Unfortunately, however, most (if not all) widely-used indexes (e.g., suffix tree, suffix array, or their compressed counterparts) are not optimized for all four measures simultaneously, as it is difficult to have the best of all four worlds. Here, we take an important step in this direction by showing that text indexing with sampling based on locally consistent anchors (lc-anchors) offers remarkably good performance in all four measures, when we have at hand a lower bound ℓ on the length of the queried patterns — which is arguably a quite reasonable assumption in practical applications. Specifically, we improve on a recently proposed index that is based on *bidirectional string anchors* (bd-anchors), a new type of lc-anchors, by: **(i)** introducing a randomized counterpart of bd-anchors which outperforms bd-anchors; **(ii)** designing an average-case linear-time algorithm to compute (the randomized) bd-anchors; and **(iii)** developing a semi-external-memory implementation and an internal-memory implementation to construct the index in *small space* using near-optimal work. Our index offers average-case guarantees. In our experiments using real (benchmark) datasets of sizes up to 10GB, we show that it compares favorably based on the four measures to all classic indexes: (compressed) suffix tree; (compressed) suffix array; and the FM-index. We also present a counterpart of our index with worst-case guarantees based on the lc-anchors notion of *partitioning sets*. To the best of our knowledge, this is the first index achieving the best of all worlds in the regime where we have at hand a lower bound ℓ on the length of the queried patterns.

Keywords Text indexing · Long patterns · Locally consistent anchors · Minimizers · Anchors

1 Introduction

In many real-world database systems, including bioinformatics systems [91], Enterprise Resource Planning (ERP) systems [86], or Business Intelligence (BI) systems [96], a large fraction of the data is represented by strings: sequences of letters over some alphabet. This is because strings can easily encode data arising from different sources such as: nucleic acid sequences read by sequencing machines (e.g., short or long DNA reads); natural language text generated by humans (e.g., description or comment fields); or identifiers generated by machines (e.g., URLs, email addresses, IP addresses). Given the ever increasing size of such datasets, it is crucial to represent them compactly [18] but also to *simultaneously* enable fast pattern matching queries. This is the classic text indexing problem [30, 52, 87]. More formally, *text indexing* asks to preprocess a string S of length n over an alphabet Σ of size σ, known as the *text*, into a compact data structure that supports efficient pattern matching queries; i.e., *decide* if a *pattern* P occurs in S or not or *report* the set of all positions in S where an occurrence of P starts.

✉ Grigorios Loukides
grigorios.loukides@kcl.ac.uk

Lorraine A. K. Ayad
lorraine.ayad@brunel.ac.uk

Solon P. Pissis
solon.pissis@cwi.nl

1 Brunel University of London, London, UK

2 King's College London, London, UK

3 CWI and Vrije Universiteit, Amsterdam, The Netherlands

1.1 Motivation and related work

A considerable amount of algorithmic research has been devoted to text indexes over the past decades [9, 10, 29, 34, 36, 43, 50, 55, 61, 71, 77, 80, 84, 97]. This is mainly due to the fact that myriad string processing tasks (see [2, 52] for comprehensive reviews) require fast access to the substrings of S. These tasks rely on such text indexes, which typically arrange the suffixes of S lexicographically in an ordered tree or in an ordered array. The former is known as the *suffix tree* [97] and the latter is known as the *suffix array* [80]. These are classic data structures, which occupy $\Theta(n)$ words of **space** (or, equivalently, $\Theta(n \log n)$ bits) and can count the number of occurrences of P in S in $\tilde{\mathcal{O}}(|P|)$ time.[1] The time for reporting is $\mathcal{O}(1)$ per occurrence for both structures. Thus, if a pattern P occurs occ times in S, the total **query time** for reporting is $\tilde{\mathcal{O}}(|P| + \mathsf{occ})$.

From early days, and in contrast to the traditional data structure literature, where the focus is on space-query time trade-offs, the main focus in text indexing has been on the **construction time**. That was until the breakthrough result of Farach [34], who showed that suffix trees (and thus suffix arrays, indirectly) can be constructed in $\mathcal{O}(n)$ time when Σ is an integer alphabet of size $\sigma = n^{\mathcal{O}(1)}$.[2] After Farach's result, more and more attention had been given to reducing the space of the index via compression techniques. This is due to the fact that, although the space is linear in the number of words, there is an $\mathcal{O}(\log_\sigma n)$ factor blowup when we consider the actual text size, which is $n\lceil \log \sigma \rceil$ bits. This factor is not negligible when σ is considerably smaller than n. For instance, the space occupied by the suffix tree of the whole human genome, even with a very efficient implementation [65] is about 40GB, whereas the genome occupies less than 1GB. To address this issue, Grossi and Vitter [50] and Ferragina and Manzini [36], and later Sadakane [92], introduced, respectively, the *compressed suffix array* (CSA), the *FM-index*, and the *compressed suffix tree* (CST). These indexes occupy $\mathcal{O}(n \log \sigma)$ bits, instead of $\mathcal{O}(n \log n)$ bits, at the expense of a factor of $\mathcal{O}(\log^\epsilon n)$ penalty in the query time, where $\epsilon > 0$ is an arbitrary predefined constant. They play a dominant role in some of the most widely used bioinformatics tools [72–74]. Mature, highly engineered implementations of these indexes are part of the sdsl-lite library of Gog et al. [45].

Nowadays, as the data volume grows rapidly, **construction space** is as well becoming crucial for several string processing tasks [10, 62]. Suffix arrays can be constructed in optimal time using $\mathcal{O}(1)$ words of extra space with the algorithm of Franceschini and Muthukrishnan [40] for general alphabets or with the algorithms of Goto [48] or Li et al. [75] for integer alphabets. *Extra* refers to the required space except for the space of S and the output. Grossi and Vitter's [50] original algorithm for constructing the CSA takes $\mathcal{O}(n \log \sigma)$ time using $\mathcal{O}(n \log n)$ bits of construction space. Hon et al. [55] showed an $\mathcal{O}(n \log \log \sigma)$-time construction reducing the construction space to $\mathcal{O}(n \log \sigma)$ bits. More recently, Belazzougui [9] and, independently, Munro et al. [84], improved the time complexity of the CSA/CST construction to $\mathcal{O}(n)$ using $\mathcal{O}(n \log \sigma)$ bits of construction space. Recently, Kempa and Kociumaka [62] have presented a new data structure that occupies $\mathcal{O}(n \log \sigma)$ bits, can be constructed in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time using $\mathcal{O}(n \log \sigma)$ bits of construction space, and has the same query time as the CSA and the FM-index.

This completes the four absolute measures anyone should pay attention to when designing or implementing a text index: space (index size); query time; construction time; and construction space. Unfortunately, however, most (if not all) widely-used indexes are not optimized for all four measures *simultaneously*, as it is difficult to have the best of all four worlds; e.g.,:

- The *suffix array* [80] supports very fast pattern matching queries; it can be constructed very fast using very little construction space; but then it occupies $\Theta(n \log n)$ bits of space and that *in any case*.
- The *CSA* [95], the *CST* [92], or the *FM-index* [36] occupy $\mathcal{O}(n \log \sigma)$ bits of space; they can be constructed very fast using $\mathcal{O}(n \log \sigma)$ bits of construction space; but then they answer pattern matching queries much less efficiently than the suffix array.

1.2 Our contributions

The purpose of this paper is to show that text indexing with *locally consistent anchors* (lc-anchors, in short) offers remarkably good performance in all four measures, when we have at hand a lower bound ℓ on the length of the queried patterns. This is arguably a quite reasonable assumption in practical applications.

One such application comes from the domain of bioinformatics, where the proposed indexing solution for exact string matching can facilitate approximate string matching [6]. In DNA sequencing [57, 76, 98], the length of sequencing reads (patterns) ranges from a few hundreds to 30 thousand [76]. The PacBio HiFi sequencing method yields highly accurate long reads of lengths averaging $10 - 25$KB and accuracy *greater than* 99.5% [54]. Suppose that a read is 20KB long. By the promised accuracy, such a long read has no more than $k = 20000 - 20000 \cdot 99.5/100 = 100$ errors. It follows by the widely-applied pigeon-hole principle that any such read can be split into $(k+1)$ fragments and *guarantee* that at least one of these fragments is matched exactly (with no error).

[1] The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses polylogarithmic factors in n.

[2] Note that the size σ of the alphabets typically used in practice are way smaller than n, and so this assumption holds.

In particular, if we divide 20KB by $k + 1 = 101$, we get fragments of length around 200. In such an application, we could thus set ℓ to 200.

Furthermore, in natural language processing, the queried patterns can also be long [94]. Examples of such patterns are queries in question answering systems [51], *description queries* in TREC datasets [3, 14], and representative phrases in documents [82]. Similarly, a query pattern can be long when it encodes an entire document (e.g., a webpage in the context of deduplication [53]), or machine-generated messages [58].

Informally, given a string S and an integer $\ell > 0$, the goal is to sample some of the positions of S (the **lc-anchors**), so as to simultaneously satisfy the following:

- *Property 1 (approximately uniform sampling):* Every fragment of length at least ℓ of S has a representative position sampled. This ensures that no pattern P, $|P| \geq \ell$, will be missed during search.
- *Property 2 (local consistency):* Exact matches between fragments of length at least ℓ of S are preserved unconditionally by having the same (relative) representative positions sampled. This ensures that similarity between similar strings of length at least ℓ will be preserved during search.

Loukides and Pissis [77] have recently introduced *bidirectional string anchors* (bd-anchors, in short), a new type of lc-anchors. The set $\mathcal{A}_\ell(S)$ of bd-anchors of order ℓ of a string S of length n is the set of starting positions of the leftmost lexicographically smallest rotation of every length-ℓ fragment of S (see Section 2 for a formal definition). The authors have shown that bd-anchors are $\mathcal{O}(n/\ell)$ in expectation [77, 78] and they can be constructed in $\mathcal{O}(n)$ worst-case time [77]. Loukides and Pissis have also proposed a text index, which is based on $\mathcal{A}_\ell(S)$ and occupies linear extra space in the size $|\mathcal{A}_\ell(S)|$ of the sample; the index can be constructed in $\tilde{\mathcal{O}}(n)$ time and can report all occ occurrences of any pattern P of length $|P| \geq \ell$ in S in $\tilde{\mathcal{O}}(|P| + \mathsf{occ})$ time (see Section 2 for a formal theorem). The authors of [77] have also implemented their index and presented some very promising experimental results (see also [78]).

We identify here two important aspects for improving the construction of the bd-anchors index: **(i)** for computing the set $\mathcal{A}_\ell(S)$, which is required to construct the index, Loukides and Pissis implement a simple $\Theta(n\ell)$-time algorithm, because their $\mathcal{O}(n)$-time worst-case algorithm seems too complicated to implement and unlikely to be efficient in practice [77]; **(ii)** their index construction implementation uses $\Theta(n)$ construction space *in any case*, which is much larger than the expected size $|\mathcal{A}_\ell(S)| = \mathcal{O}(n/\ell)$ of the index.

Here, we improve on the index proposed by Loukides and Pissis [77] by addressing these aspects as follows:

- We introduce the notion of randomized bd-anchors; the randomized counterpart of bd-anchors. Informally, the set $\mathcal{A}^h_\ell(S)$ of *randomized bd-anchors* of order ℓ of a string S of length n is the set of starting positions of the leftmost "smallest" rotation of every length-ℓ fragment of S (see Section 4). The order of rotations is defined based on a ranking function h *and*, additionally, on the standard lexicographic order. We show that randomized bd-anchors are $\mathcal{O}(n/\ell)$ in expectation when the order is based on a uniformly random permutation.
- We design a novel average-case $\mathcal{O}(n)$-time algorithm to compute either $\mathcal{A}_\ell(S)$ or $\mathcal{A}^h_\ell(S)$.[3] To achieve this, we use minimizers [90, 93], another well-known type of lc-anchors, as anchors to compute $\mathcal{A}_\ell(S)$ (or $\mathcal{A}^h_\ell(S)$) after carefully setting the sampling parameters. We employ longest common extension (LCE) queries [61] on S to compare anchored rotations (i.e., rotations of S starting at minimizers) of fragments of S efficiently. The fact that minimizers are $\mathcal{O}(n/\ell)$ in expectation lets us realize the average-case $\mathcal{O}(n)$-time computation of $\mathcal{A}_\ell(S)$ (or $\mathcal{A}^h_\ell(S)$).
- We propose a semi-external-memory and an internal-memory implementation to construct the index in small space using near-optimal work when the set of bd-anchors is given. We first compute $\mathcal{A}_\ell(S)$ using $\mathcal{O}(\ell)$ space using the aforementioned algorithm. For the semi-external-memory implementation, we show that if we have $\mathcal{A}_\ell(S)$ in internal memory and the suffix array of S in external memory, it suffices to scan the suffix array sequentially to deduce the information required to construct the index thus using $\mathcal{O}(\ell + |\mathcal{A}_\ell(S)|)$ space. For the internal-memory implementation, we use *sparse suffix sorting* [5] on $\mathcal{A}_\ell(S)$ to construct the index in $\mathcal{O}(\ell + |\mathcal{A}_\ell(S)|)$ space.
- We present an extensive experimental evaluation using: (1) real benchmark datasets [35]; (2) real large datasets of sizes up to 10GB; and (3) data based on a use case. For (1), we first show that our new algorithm for computing $\mathcal{A}_\ell(S)$ (or $\mathcal{A}^h_\ell(S)$) is *more than two orders of magnitude faster* as ℓ increases than the simple $\Theta(n\ell)$-time algorithm, while using a very similar amount of memory. We also show that randomized bd-anchors outperform bd-anchors *in all measures*: construction time, construction space, and sample size. We remark that (instead of a uniformly random order) we used Karp-Rabin (KR) fingerprints [59] to implement function h. We then go on to examine the index construction based on the above four measures. The results show that, for long patterns, the index constructed using randomized bd-anchors and

[3] The algorithm works for a reduced version of $\mathcal{A}_\ell(S)$ or $\mathcal{A}^h_\ell(S)$ [78]. For simplicity, we avoid making this version explicit in Introduction; see Section 2 for a formal definition.

our improved construction algorithms compares favorably to all classic indexes (implemented using sdsl-lite): (compressed) suffix tree; (compressed) suffix array; and the FM-index. For instance, our index offers about 27% faster query time, for all datasets on average, for all $\ell \in [32, 1024]$, compared to the suffix array (which performs best among the competitors in this measure), while occupying up to two orders of magnitude less space. Also, our index occupies up to 5× less space on average over all datasets, for $\ell = 2^{10}$, compared to the FM-index (which performs best among the competitors in this measure), while being up to one order of magnitude faster in query time. For (2), we show similar results. For example, our index on the full human genome occupies 11MB for $\ell = 2^{14}$ and answers queries more than 72× faster than the FM-index, which occupies more than 1GB. As another example, our index on a repetitive dataset occupies 42MB for $\ell = 2^{14}$ and answers queries more than 12 times faster than the r-index [43], which occupies 400MB. This is encouraging as the r-index is designed for repetitive data. For (3), we demonstrate the benefits of our index in a realistic use case of mapping PacBio HiFi reads.

– The above algorithms are based on an *average-case* analysis. In the worst case, however, $|\mathcal{A}_\ell(S)|$ (or $|\mathcal{A}^h_\ell(S)|$) are in $\Theta(n)$. This implies that our index does not offer satisfactory worst-case guarantees. Motivated by this, we also present a counterpart of our index with *worst-case* guarantees based on the lc-anchors notion of *partitioning sets* [64]. This construction is mainly of theoretical interest. However, we stress that, to the best of our knowledge, it is the first index achieving the best of all worlds in the regime where we have at hand a lower bound ℓ on the length of the queried patterns. Specifically, this index occupies $\mathcal{O}(n/\ell)$ space, it can be constructed in near-optimal time using $\mathcal{O}(n/\ell)$ space, and it supports near-optimal pattern matching queries. With *near-optimal*, we mean that the construction time is $\tilde{\mathcal{O}}(n)$ and the query time is $\tilde{\mathcal{O}}(|P| + \mathsf{occ})$; i.e., we neglect the factors that are polylogarithmic in n and aim at improvements in space that are *linear in* ℓ.

1.3 Other related work

The connection of lc-anchors to text indexing and other string-processing applications is not new. The perhaps most popular lc-anchors in practical applications are *minimizers*, which have been introduced independently by Schleimer et al. [93] and by Roberts et al. [90] (see Section 2 for a definition). Although minimizers have been mainly used for sequence comparison [57], Grabowski and Raniszewski [49] showed how minimizers can be used to sample the suffix array – see also [28], which uses an alphabet sampling approach. Another well-known notion of lc-anchors is *difference covers*, which have been introduced by Burkhardt and Kärkkäinen [22] for suffix array construction in small space (see also [79]). Difference covers play also a central role in the elegant linear-time suffix array construction algorithm of Kärkkäinen et al. [71]; and they have been used in other string processing applications [13, 24]. Another very powerful type of lc-anchors is *string synchronizing sets*, which have been recently proposed by Kempa and Kociumaka [61], for constructing, among others, an optimal data structure for LCE queries (see also [33]). String synchronizing sets have applications in designing sublinear-time algorithms for text indexing [62] and for other classic string problems [25, 27], whose textbook linear-time solutions rely on suffix trees or arrays. Another type of lc-anchors, closely related to string synchronizing sets, is *partitioning sets* [17, 64].

1.4 Paper organization and conference version

In Section 2, we present the necessary definitions and notation, a brief overview of classic text indexes, as well as some existing results on minimizers and bd-anchors. In Section 3, we present a brief overview of the index proposed by Loukides and Pissis [77]. In Section 4, we describe randomized bd-anchors, our new sampling mechanism, in detail. In Section 5, we improve the construction of the index of Section 3 by presenting: **(i)** our fast algorithm for (randomized) bd-anchors computation (see Section 5.1); and **(ii)** the index construction in small space using near-optimal work (see Section 5.2). In Section 6, we present the counterpart of our index offering worst-case guarantees. In Section 7, we provide the full details of our implementations, and in Section 8, we present our extensive experimental evaluation. We conclude this paper in Section 9.

A preliminary version of this paper was announced at PVLDB 2023 [4]. Therein we identified two ways for potentially improving the *construction time* of the index: **(i)** design a new sampling mechanism that reduces the number of minimizers to potentially improve the construction of bd-anchors; **(ii)** plug sparse suffix sorting in the semi-external-memory implementation to make it work fully in internal memory thus potentially improving the construction of the index. In response (to **(i)**), we introduce here the notion of randomized bd-anchors (see Section 4). It is well-known that randomized minimizers have usually a lower density than their lexicographic counterparts [99]; in practice, the former are typically implemented using a rolling hash function (e.g., KR fingerprints), which is also faster and more space-efficient. Unfortunately, for bd-anchors, it does not make sense to maintain the hash value of the rotations in every window. This is because when shifting the window, the hash value of each rotation changes and thus *the minimizer position will likely change as well*. We thus partition the rotations in two parts: the first one is of length $r + 1$; and the second one is of length $\ell - (r + 1)$. We compute the *hash value* of the former and the *lexicographic rank* of the latter. These two integers form a pair. The rotation sampled is the rotation with the smallest such pair (in lexicographic order). This lets us *simultaneously* exploit the benefits of randomized minimizers (small sample, small space, and fast computation) and bd-anchors (effective tie-breaking). For **(ii)**, we plug the sparse suffix sorting implementation of Ayad et al. [5] in our semi-external-memory implementation (see Section 5).

Since our new implementations (see Section 7) based on the aforementioned two changes (randomized bd-anchors and internal-memory implementation) have dramatically improved the construction in the preliminary version [4] (which was based on lexicographic bd-anchors and on external memory), *the experimental section is also completely new* (see Section 8): we have re-conducted all experiments from scratch; we have re-plotted the results, and a new description has been added. Furthermore, we now use two new large datasets and present experimental results using PacBio HiFi reads.

Section 6, presenting the counterpart of our index with worst-case guarantees, is also completely new.

2 Preliminaries

An *alphabet* Σ is a finite set of elements called *letters*; we denote by σ the size $|\Sigma|$ of Σ. A *string* $S = S[1 \mathinner{.\,.} n]$ is a sequence of letters over some alphabet Σ; we denote by $|S| = n$ the length of S. The fragment $S[i \mathinner{.\,.} j]$ of S is an *occurrence* of the underlying *substring* $P = S[i] \ldots S[j]$. We also write that P occurs at *position* i in S when $P = S[i] \ldots S[j]$. A substring P of S is called *proper* when $P \neq S$. A *prefix* of S is a fragment of S of the form $S[1 \mathinner{.\,.} j]$ and a *suffix* of S is a fragment of S of the form $S[i \mathinner{.\,.} n]$. Given a string S and an integer $1 \leq i \leq |S|$, we define $S[i \mathinner{.\,.} |S|]S[1 \mathinner{.\,.} i-1]$ to be the *ith rotation* of S. Given a string S of length n, we denote by $\overleftarrow{S}$ the *reverse* $S[n] \ldots S[1]$ of S.

TEXT INDEXING
Preprocess: A string S of length n over an integer alphabet of size $\sigma = n^{\mathcal{O}(1)}$.
Query: Given a string P of length m, report all positions i such that $P = S[i \mathinner{.\,.} i + m - 1]$.

Classic text indexes

For a string S of length n over an ordered alphabet of size σ, the *suffix array* $\mathsf{SA}[1 \mathinner{.\,.} n]$ stores the permutation of $\{1, \ldots, n\}$ such that $\mathsf{SA}[i]$ is the starting position of the ith lexicographically smallest suffix of S. The standard application of SA is text indexing, in which we consider S to be the *text*: given any string $P[1 \mathinner{.\,.} m]$, known as the *pattern*, the suffix array of S allows us to report all occ occurrences of P in S using only $\mathcal{O}(m \log n + \mathsf{occ})$ operations [80]. We perform binary search in SA resulting in a range $[s, e)$ of suffixes of S having P as a prefix. Then, $\mathsf{SA}[s \mathinner{.\,.} e-1]$ contains the starting positions of all occurrences of P in S. The SA is often augmented with the LCP array [80] storing the length of longest common prefixes of lexicographically adjacent suffixes (i.e., consecutive entries in the SA). In this case, reporting all occ occurrences of P in S can be done in $\mathcal{O}(m + \log n + \mathsf{occ})$ time by avoiding to compare P with suffixes of S from scratch during binary search [80] (see [29, 38, 88] for subsequent improvements). The suffix array occupies $\Theta(n)$ space and it can be constructed in $\mathcal{O}(n)$ time, when $\sigma = n^{\mathcal{O}(1)}$, using $\mathcal{O}(n)$ space [34]. Given SA of S, we can compute the LCP array of S in $\mathcal{O}(n)$ time [60].

Given a set $\mathcal{F}$ of strings, the *compacted trie* of these strings is the trie obtained by compressing each path of nodes of degree one in the trie of the strings in $\mathcal{F}$, which takes $\mathcal{O}(|\mathcal{F}|)$ space [83]. Each edge in the compacted trie has a label represented as a fragment of a string in $\mathcal{F}$. The *suffix tree* of S, which we denote by $\mathsf{ST}(S)$, is the compacted trie of the suffixes of S [97]. Assuming S ends with a unique terminating symbol, every leaf in $\mathsf{ST}(S)$ represents a suffix $S[i \mathinner{.\,.} n]$ and is decorated by index i. The set of indices stored at the leaf nodes in the subtree rooted at node v is the *leaf-list* of v, and we denote it by $LL(v)$. Each edge in $\mathsf{ST}(S)$ is labelled with a nonempty substring of S such that the path from the root to the leaf annotated with index i spells the suffix $S[i \mathinner{.\,.} n]$. The substring of S spelled by the path from the root to node v is the *path-label* of v, and we denote it by $L(v)$. Given any pattern $P[1 \mathinner{.\,.} m]$, $\mathsf{ST}(S)$ allows us to report all occ occurrences of P in S using only $\mathcal{O}(m \log \sigma + \mathsf{occ})$ operations. We simply spell P from the root of $\mathsf{ST}(S)$ (to access edges by the first letter of their label, we use binary search) until we arrive (if possible) at the first node v such that P is a prefix of $L(v)$. Then all occ occurrences (starting positions) of P in S are $LL(v)$. The suffix tree occupies $\Theta(n)$ space and it can be constructed in $\mathcal{O}(n)$ time, when $\sigma = n^{\mathcal{O}(1)}$, using $\mathcal{O}(n)$ space [34]. To improve the query time to the optimal $\mathcal{O}(m + \mathsf{occ})$ we can use randomization to construct a perfect hash table [41] to access edges by the first letter of their label in $\mathcal{O}(1)$ time; see also [38] for deterministic improvements in query time.

Let S be a string of length n. Given two integers $1 \leq i, j \leq n$, we denote by $\mathsf{LCP}_S(i, j)$ the length of the longest common prefix (LCP) of $S[i \mathinner{.\,.} n]$ and $S[j \mathinner{.\,.} n]$. When S is over an integer alphabet of size $\sigma = n^{\mathcal{O}(1)}$, we can construct a data structure in $\mathcal{O}(n/\log_\sigma n)$ time that answers $\mathsf{LCP}_S(i, j)$ queries in $\mathcal{O}(1)$ time [61].

Lc-anchors

Given a string S of length n, two integers $w, k > 0$, and the ith length-$(w+k-1)$ fragment $F = S[i \mathinner{.\,.} i+w+k-2]$ of S, the (w, k)-*minimizers* of F are defined as the positions

$j \in [i, i+w)$ where a lexicographically minimal length-k substring of F occurs [90]. The set $\mathcal{M}_{w,k}(S)$ of (w,k)-minimizers of S is defined as the set of (w,k)-minimizers of each fragment $S[i\,..\,i+w+k-2]$, for all $i \in [1, n-w-k+2]$.

Example 1 (Minimizers) Let $S = \texttt{aacaaacgcta}$ and $w = k = 3$. We consider fragments of length $w+k-1 = 5$. The first fragment is aacaa. The lexicographically minimal length-k substring is aac, starting at position 1, so we add position 1 to $\mathcal{M}_{3,3}(S)$. The second fragment is acaaa. The lexicographically minimal length-k substring is aaa, starting at position 4, so we add position 4 to $\mathcal{M}_{3,3}(S)$. The third fragment, caaac, and fourth fragment, aaacg, have aaa as the lexicographically minimal length-k substring, so $\mathcal{M}_{3,3}(S)$ does not change. The fifth fragment is aacgc. The lexicographically minimal length-k substring is aac, starting at position 5, and so we add position 5 to $\mathcal{M}_{3,3}(S)$. The sixth fragment is acgct. The lexicographically minimal length-k substring is acg, starting at position 6, and so we add position 6 to $\mathcal{M}_{3,3}(S)$. The seventh fragment is cgcta. The lexicographically minimal length-k substring is cgc, starting at position 7, and so we add position 7 to $\mathcal{M}_{3,3}(S)$. Thus $\mathcal{M}_{3,3}(S) = \{1, 4, 5, 6, 7\}$.

Lemma 1 *([99]) If S is a string of length n, randomly generated by a memoryless source over an alphabet of size $\sigma \geq 2$ with identical letter probabilities, then the expected size of $\mathcal{M}_{w,k}(S)$ is $\mathcal{O}(n/w)$ if and only if $k \geq \log_\sigma w + \mathcal{O}(1)$.*

Lemma 2 *([77]) For any string S of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$ and integers $w, k > 0$, the set $\mathcal{M}_{w,k}$ can be computed in $\mathcal{O}(n)$ time.*

Loukides and Pissis defined the following alternative notion of lc-anchors [77].

Definition 1 (Bidirectional anchor) Given a string F of length $\ell > 0$, the *bidirectional anchor* (bd-anchor) of F is the lexicographically minimal rotation $j \in [1, \ell]$ of F with minimal j. The set of order-ℓ bd-anchors of a string S of length $n > \ell$, for some integer $\ell > 0$, is defined as the set $\mathcal{A}_\ell(S)$ of bd-anchors of $S[i\,..\,i+\ell-1]$, for all $i \in [1, n-\ell+1]$. In particular, if j is the bd-anchor of $S[i\,..\,i+\ell-1]$, then $i+j-1$ is added to $\mathcal{A}_\ell(S)$.

Example 2 (Bd-anchors) Let $S = \texttt{aacaaacgcta}$ and $\ell = 5$. We consider fragments of length $\ell = 5$. The first fragment is aacaa. We need to consider all of its rotations and select the leftmost lexicographically minimal. All rotations of aacaa are: aacaa, acaaa, caaaa, aaaac, and aaaca. We thus select aaaac, which starts at position 4, and add position 4 to $\mathcal{A}_5(S)$. The second fragment is acaaa. All rotations of acaaa are: acaaa, caaaa, aaaac, aaaca, and aacaa. We thus select aaaac, which starts at position 4, and so we do not need to add position 4 to $\mathcal{A}_5(S)$. The third fragment is caaac. All rotations of caaac are: caaac, aaacc, aacca, accaa, and ccaaa. We thus select aaacc, which starts at position 4, and so we do not need to add position 4 to $\mathcal{A}_5(S)$. The fourth fragment is aaacg, which is also the lexicographically minimal rotation. This rotation starts at position 4, and so we do not need to add position 4 to $\mathcal{A}_5(S)$. The fifth fragment is aacgc, which is also the lexicographically minimal rotation. This rotation starts at position 5, and so we add position 5 to $\mathcal{A}_5(S)$. The sixth fragment is acgct, which is also the lexicographically minimal rotation. This rotation starts at position 6, and so we add position 6 to $\mathcal{A}_5(S)$. The seventh fragment is cgcta. The lexicographically minimal rotation of this fragment is acgct, which starts at position 11, and so we add position 11 to $\mathcal{A}_5(S)$. Thus $\mathcal{A}_5(S) = \{4, 5, 6, 11\}$.

The bd-anchors notion was also parameterized by Loukides et al. [78] according to the following definition.

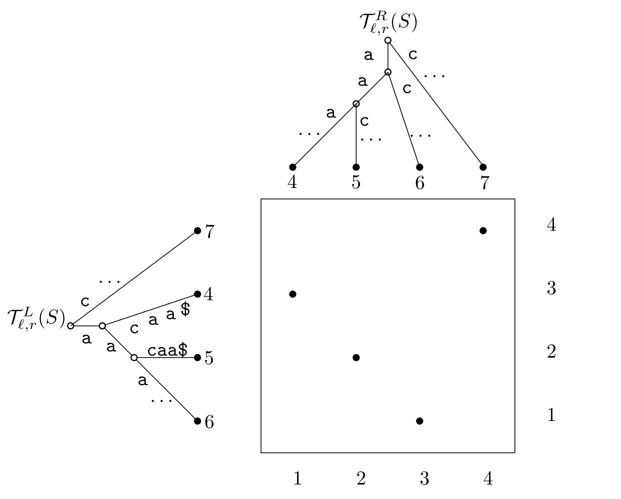
Definition 2 (Reduced bd-anchor) Given a string F of length $\ell > 0$ and an integer $0 \leq r \leq \ell - 1$, we define the *reduced bidirectional anchor* of F as the lexicographically minimal rotation $j \in [1, \ell-r]$ of F with minimal j. The set of order-ℓ reduced bd-anchors of a string S of length $n > \ell$ is defined as the set $\mathcal{A}_{\ell,r}(S)$ of reduced bd-anchors of $S[i\,..\,i+\ell-1]$, for all $i \in [1, n-\ell+1]$. In particular, if j is the reduced bd-anchor of $S[i\,..\,i+\ell-1]$, then $i+j-1$ is added to $\mathcal{A}_{\ell,r}(S)$.

Informally, the reduced bd-anchor mechanism neglects the r rightmost rotations in the sampling process.

Example 3 (Reduced bd-anchors) Let $S = \texttt{aacaaacgcta}$, $\ell = 5$, and $r = 1$. Recall from Example 2 that $\mathcal{A}_5(S) = \{4, 5, 6, 11\}$. To see the difference between bd-anchors and reduced bd-anchors, consider the seventh (last) fragment of length $\ell = 5$ of S. This fragment is cgcta and its rotations are: cgcta, gctac, ctacg, tacgc, and acgct. The lexicographically minimal rotation is acgct and this is why $11 \in \mathcal{A}_5(S)$. In the case of reduced bd-anchors we are asked to neglect the $r = 1$ rightmost rotations, and so the only candidates are: cgcta, gctac, ctacg, and tacgc. Out of these, the lexicographically minimal rotation is cgcta, which starts at position 7, and this is why $\mathcal{A}_{5,1}(S) = \{4, 5, 6, 7\}$.

Lemma 3 *([77, 78]) If S is a string of length n, randomly generated by a memoryless source over an alphabet of size $\sigma \geq 2$ with identical letter probabilities, then, for any integer $\ell > 0$, the expected size of $\mathcal{A}_{\ell,r}(S)$ with $r = \lceil 4 \log \ell / \log \sigma \rceil$ is in $\mathcal{O}(n/\ell)$.*

Lemma 4 *([77, 78]) For any string S of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$ and integers $\ell, r > 0$, the set $\mathcal{A}_{\ell,r}(S)$ can be computed in $\mathcal{O}(n)$ time.*


(a) The $\mathcal{I}_{\ell,r}(S)$ index for $S=$ aacaaacgcta, $\ell=5$, and $r=1$.

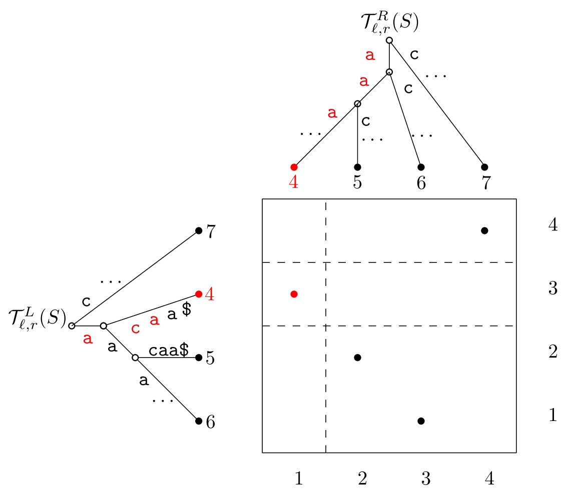

(b) Querying $P=$ acaaa.

Fig. 1 (See Example 2). Let $S=$ aacaaacgcta, $\ell=5$, and $r=1$. We have that $\mathcal{A}_{5,1}(S)=\{4,5,6,7\}$, and thus the indexed suffixes and reversed suffixes of S are as shown in (a). Assume that we have a query pattern $P=$ acaaa. We first find the reduced bd-anchor of $P[1\,..\,\ell]$ for $\ell=5$ and $r=1$, which is $j=3$: the minimal lexicographic rotation is aaaac. We then split P in $\overleftarrow{P[1\,..\,3]}=$ aca and $P[3\,..\,|P|]=$ aaa, and search $\overleftarrow{P[1\,..\,3]}=$ aca in $\mathcal{T}^L_{\ell,r}$ and $P[3\,..\,|P|]=$ aaa in $\mathcal{T}^R_{\ell,r}$. The search induces a rectangle, which encloses all the occurrences of P in S, as shown in (b). Indeed, $P=S[2\,..\,6]=$ acaaa: the fragment $S[2\,..\,6]$ is anchored at position 4

3 The index

Loukides and Pissis [77] proposed the following text index, which is based on (reduced [78]) bd-anchors. Note that the same index can be constructed for any lc-anchors (e.g., for minimizers) satisfying Property 1 (approximately uniform sampling) and Property 2 (local consistency). Property 1 requires that we have one position sampled for every length-ℓ fragment F of S; and Property 2 requires that the sampling criteria depend solely on F. Let us denote the sample by $\mathcal{A}(S)$.

Fix string S of length n. Given $\mathcal{A}(S)$, we construct two compacted tries: one for strings $S[i\,..\,n]$, for all $i\in\mathcal{A}(S)$, which we denote by $\mathcal{T}^R(S)$; and one for strings $\overleftarrow{S[1\,..\,i]}$, for all $i\in\mathcal{A}(S)$, which we denote by $\mathcal{T}^L(S)$. We also construct a 2D range reporting data structure [23] over $|\mathcal{A}(S)|$ points (x,y), where x is the lexicographic rank of $S[i\,..\,n]$ and y is the lexicographic rank of $\overleftarrow{S[1\,..\,i]}$, for all $i\in\mathcal{A}(S)$. For any query P of length $|P|\geq\ell$, we find the lc-anchor of $P[1\,..\,\ell]$, say j, which gives $P[j\,..\,|P|]$ and $\overleftarrow{P[1\,..\,j]}$. We search for the former in $\mathcal{T}^R$ and for the latter in $\mathcal{T}^L$. This search induces a rectangle, which we use to query the 2D range reporting data structure. The reported points are all the occurrences of P in S and only those.

Let us set $\mathcal{A}(S):=\mathcal{A}_{\ell,r}(S)$ and denote the resulting index by $\mathcal{I}_{\ell,r}(S)$ (inspect Figure 1, for an example). Loukides and Pissis showed the following result [77, 78].

Theorem 1 *([77, 78]) Given any string S of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$ and integers $\ell,r>0$, the index $\mathcal{I}_{\ell,r}(S)$ occupies $\mathcal{O}(|\mathcal{A}_{\ell,r}(S)|)$ extra space and reports all* occ *occurrences of any pattern P of length $|P|\geq\ell$ in S in $\tilde{\mathcal{O}}(|P|+\mathsf{occ})$ time. Moreover, the index $\mathcal{I}_{\ell,r}(S)$ can be constructed in $\tilde{\mathcal{O}}(n)$ time.*

Extra refers to the $n\lceil\log\sigma\rceil$ bits required to store S. Alternatively one could use $nH_0(S)+o(nH_0(S))+o(n)$ bits without any penalty [7, 11], where $H_0(S)$ is the *zeroth-order entropy* of S; or in fact any other small structure supporting $\tilde{\mathcal{O}}(1)$-time random access queries.

4 Randomized reduced bidirectional anchors

We introduce the notion of randomized reduced bd-anchors by injecting randomization to the notion of reduced bd-anchors [78]. This is analogous to the idea of using randomization to select minimizers [93] instead of using the standard lexicographic total order [90]. As is common in the literature (cf. [99]), we will henceforth assume a *random order* $h:\Sigma^k\rightarrow\mathbb{N}$, that is, a permutation of Σ^k uniformly chosen over all possible permutations. Implementing a randomized minimizers mechanism in practice is straightforward via using a (pseudo)random hash function h (e.g., KR fingerprints). The other benefit of randomized minimizers is that they usually have lower density than their lexicographic counterparts [99].

Unfortunately, it is not as straightforward to inject randomization to (reduced) bd-anchors. Intuitively, this is because bd-anchors wrap around the window. We next explain how this can be achieved by using a mix of lexicographic total order and a random order. Importantly, our experiments show a clear superiority (lower density) of this new randomized notion over the deterministic (lexicographic) counterpart.

The definition of randomized reduced bd-anchors (Definition 3) is a bit involved. We will thus start with some intuition. Given ℓ and r (exactly as in the reduced bd-anchors case) and a fragment F of S, we select as *candidates*, the $(\ell-r, r+1)$-minimizers of F based on the total order induced by the random order. If there exists only one candidate selected, then this is the randomized reduced bidirectional anchor of F. If there are many candidates, then we select the leftmost lexicographically smallest rotation out of all the rotations starting right after the last position of the candidates. The candidate whose rotation is selected is then selected as the randomized reduced bidirectional anchor of F. We next formalize this intuition.

Definition 3 (Randomized reduced bd-anchor) Given a string F of length $\ell > 0$, an integer $0 \leq r \leq \ell - 1$, and a ranking function $h : \Sigma^{r+1} \rightarrow \mathbb{N}$, we define the *randomized reduced bidirectional anchor* of F as the starting position of the length-$(r+1)$ fragment of F with the minimal rank in h. If there are $e > 1$ such fragments $F[i_1 \mathinner{..} j_1], \dots, F[i_e \mathinner{..} j_e]$ in F (i.e., e fragments that share the same minimal rank in h), then we resolve the ties by choosing $j - (r+1)$ as the randomized reduced bd-anchor of F, where j is the lexicographically minimal rotation from $\{j_1 + 1, \dots, j_e + 1\}$ with minimal j. The set of order-ℓ randomized reduced bd-anchors of a string S of length $n > \ell$ is defined as the set $\mathcal{A}^h_{\ell,r}(S)$ of randomized reduced bd-anchors of $S[i \mathinner{..} i+\ell-1]$, for all $i \in [1, n-\ell+1]$. In particular, if j is the randomized reduced bd-anchor of $S[i \mathinner{..} i+\ell-1]$, then $i+j-1$ is added to $\mathcal{A}_{\ell,r}(S)$.

Example 4 (Randomized reduced bd-anchor)
Let $F = \texttt{aacaaacgcta}$ of length $\ell = 11$ and $r = 2$. We need to consider the following set of length-3 substrings of F:

$$\{\texttt{aac}, \texttt{aca}, \texttt{caa}, \texttt{aaa}, \texttt{acg}, \texttt{cgc}, \texttt{gct}, \texttt{cta}\}.$$

Assume that $h(F[1 \mathinner{..} 3])$, that is, the rank of $F[1 \mathinner{..} 3] = F[5 \mathinner{..} 7] = \texttt{aac}$, is the smallest one among the ranks of all length-3 substrings of F. We have that $e = 2$ because $\texttt{aac}$ occurs at two positions: 1 and 5. To resolve the tie, we need to consider the rotations starting at positions in $\{4, 8\}$. These are: $\texttt{aaacgctaaac}$ and $\texttt{gctaaacaaac}$. The former $(j = 4)$ is the lexicographically smallest, and so we select $j - (r+1) = 4 - (2+1) = 1$ as the randomized reduced bd-anchor of F.

Randomized reduced bd-anchors are a new type of lc-anchors (see Section 2). Indeed, like minimizers [100] and (reduced) bd-anchors [78], it is straightforward to prove that randomized reduced bd-anchor samples enjoy the following two useful properties (see Section 1):

- *Property 1 (approximately uniform sampling)*: By the definition of randomized reduced bd-anchors (see Definition 3), every fragment of length at least ℓ of S has a position added to set $\mathcal{A}^h_{\ell,r}(S)$.
- *Property 2 (local consistency)*: Again, by the definition of randomized reduced bd-anchors (see Definition 3), the selection criteria for the length-$(r+1)$ fragment of F depend only on F; and then, in particular, the tie-breaking rule is also only dependent on F, and nothing outside F. Thus, if two strings X and Y share F as a substring, the position selected within F will appear in both $\mathcal{A}^h_{\ell,r}(X)$ and $\mathcal{A}^h_{\ell,r}(Y)$, which can then be used to recover the match.

We denote by $\mathcal{M}^h_{w,k}(S)$ the set of (w,k)-minimizers of S when the underlying total order on length-k substrings is determined by a ranking function h. In particular, the following lemma is known when h is a uniformly random permutation.

Lemma 5 *([99]) If S is a string of length n, randomly generated by a memoryless source over an alphabet of size $\sigma \geq 2$ with identical letter probabilities, then the expected size of $\mathcal{M}^h_{w,k}(S)$ with $k > (3+\epsilon)\log_\sigma(w+1)$ is in $\mathcal{O}(n/w)$, for any $\epsilon > 0$, when h is a uniformly random permutation.*

The following lemma follows directly from Lemma 5 and the fact that by construction $|\mathcal{A}^h_{\ell,r}(S)| \leq |\mathcal{M}^h_{w,k}(S)|$, where $k = r+1$ and $w = \ell - r$.

Lemma 6 *If S is a string of length n, randomly generated by a memoryless source over an alphabet of size $\sigma \geq 2$ with identical letter probabilities, then, for any integer $\ell > 0$, the expected size of $\mathcal{A}^h_{\ell,r}(S)$ with $r = \lceil 4\log\ell/\log\sigma \rceil$ is in $\mathcal{O}(n/\ell)$, when h is a uniformly random permutation.*

Comparing Lemma 6 with Lemma 3, we see that this new notion of bd-anchors has the same expected size in the worst case as reduced bd-anchors. However, as mentioned before, our experiments show a clear superiority (lower density) of this new randomized notion over the deterministic (lexicographic) counterpart. In the next section, we show that both notions can be computed in linear time with the same technique. For randomized bd-anchors, we will assume that a random order is given. Before proceeding, we give a standard practical implementation for a (pseudo)random order.

Implementing a (pseudo)random order To implement a (pseudo)random order in practice, we use KR fingerprints. Let S be a string of length n over an integer alphabet. Let p be a prime and choose $r \in [0, p-1]$ uniformly at random.

The KR fingerprint of $S[i\,..\,j]$ is defined as:

$$\phi_S(i,j) = \sum_{k=i}^{j} S[k]r^{j-k} \mod p.$$

Clearly, if $S[i\,..\,i+\ell] = S[j\,..\,j+\ell]$ then $\phi_S(i,i+\ell) = \phi_S(j,j+\ell)$. On the other hand, if $S[i\,..\,i+\ell] \neq S[j\,..\,j+\ell]$ then $\phi_S(i,i+\ell) \neq \phi_S(j,j+\ell)$ with probability at least $1-\ell/p$ [31]. Since we are comparing only substrings of equal length, the number of different possible substring comparisons is less than n^3. Thus, for any constant $c \geq 1$, we can set p to be a prime larger than $\max(|\Sigma|, n^{c+3})$ to make the KR fingerprint function perfect (i.e., no collisions) with probability at least $1-n^{-c}$ (this means *with high probability*).[4] Any KR fingerprint of S or p fits in one machine word of $\Theta(\log n)$ bits. In our methods, the length $j-i+1$ of the substrings of S considered will be fixed, hence the quantity $r^{j-i+1} \mod p$ needs to be computed only once for the KR fingerprint computations in a sliding window on S.

5 Improving the index

In this section, we improve the construction of the index $\mathcal{I}_{\ell,r}(S)$ (Theorem 1) focusing on two aspects: **(i)** the fast computation of (randomized) reduced bd-anchors (see Section 5.1); and **(ii)** the index construction in small space using near-optimal work (see Section 5.2). With *small*, we mean space close to the index size.

5.1 Computing bd-anchors in linear time

Recall that the first step to construct the $\mathcal{I}_{\ell,r}(S)$ index is to compute the set $\mathcal{A}_{\ell,r}(S)$ of reduced bd-anchors of S. Loukides and Pissis [77, 78] showed a worst-case linear-time algorithm to compute $\mathcal{A}_{\ell,r}(S)$ (Lemma 4). Although this algorithm is optimal in the worst case, it seems quite complex to implement and it is unlikely to be efficient in practice. The worst-case linear-time algorithm for computing $\mathcal{A}_{\ell,r}(S)$ relies on a data structure introduced by Kociumaka in [63]. The latter data structure relies heavily on the construction of *fusion trees* [42], and hence it is mostly of theoretical interest: it is widely accepted that naive solutions can be more practical than such sophisticated data structures – see [32] and references therein. That is why Loukides and Pissis have instead proposed and implemented a simple $\Theta(n\ell)$-time algorithm to compute $\mathcal{A}_{\ell,r}(S)$ in their experiments [77]. Here we give a novel average-case linear-time algorithm to compute $\mathcal{A}_{\ell,r}(S)$. In the worst case, the algorithm runs in $\mathcal{O}(n\ell)$ time as the simple algorithm from [77] does.

[4] Note that in our datasets, where n is in the order of millions or billions, the probability of an error is completely negligible.

Let us remark that the same technique (with very small differences) can be used to compute $\mathcal{A}^h_{\ell,r}(S)$, the set of randomized reduced bd-anchors of S. We will highlight these differences at the end of this section.

Main idea We use (w,k)-minimizers as anchors to compute reduced bd-anchors of order ℓ after setting parameters w and k. We employ $\mathsf{LCP}_S(i,j)$ queries to compare anchored rotations (i.e., rotations of S starting at (w,k)-minimizers) of fragments of S efficiently. The fact that (w,k)-minimizers are $\mathcal{O}(n/\ell)$ in expectation lets us realize $\mathcal{O}(n)$ time on average.

Let us start with the following simple fact.

Fact 1 *For any string F of length $\ell > 0$, the reduced bd-anchor of F, for any $1 \leq r \leq \ell-1$, is an $(\ell-r, r+1)$-minimizer of F.*

Proof Let j be the reduced bd-anchor of F: the lexicographically smallest rotation of F with minimal $j \in [1, \ell-r]$. By definition, $F[j\,..\,|F|]$ is of length at least $r+1$ and there cannot be another j' such that $F[j'\,..\,j'+r]$ is lexicographically smaller than $F[j\,..\,j+r]$. Since $\ell = w+k-1 \implies w = \ell - r$, j is an $(\ell-r, r+1)$-minimizer of F. □

Example 5 (Cont'd from Example 3) The set $\mathcal{M}_{w,k}(S)$ of minimizers for $w = \ell - r = 4$ and $k = r+1 = 2$ is $\mathcal{M}_{4,2}(S) = \{1,4,5,6,7\}$. Since any reduced bd-anchor for $\ell = 5$ and $r = 1$ is a $(4,2)$-minimizer of S, $\mathcal{A}_{5,1}(S) = \{4,5,6,7\} \subseteq \mathcal{M}_{4,2}(S)$.

In particular, Fact 1 implies that, for any string S of length n and integers $w, k > 0$, the set of (w,k)-minimizers in S is a superset of the set of reduced bd-anchors of order $\ell = w+k-1$ with $r = k-1$. Thus, we will use (w,k)-minimizers (computed by means of Lemma 2) to compute reduced bd-anchors by setting $w = \ell - r$ and $k = r+1$. We remark that, unlike Lemma 4 (see above), Lemma 2 is extremely efficient in practice and is simple to implement, as it relies on suffix arrays and a single left-to-right pass using a deque (e.g., `std::deque` in `C++`) to maintain the minimum.

The following lemma is crucial for the efficiency of our algorithm. The lemma is not new; it has previously been used; see, for instance, [63]. The main idea is to use LCP queries to quickly identify suitable substrings (of the input rotations) that are then compared, in order to determine the lexicographically smallest rotation.

Lemma 7 *For any string F and two positions i and j on F, we can determine which of the rotations i or j is the smaller lexicographically in the time to answer three LCP_F queries and three letter comparisons in F.*

Proof Assume without loss of generality that $i < j$ (inspect Figure 2). We first ask for the length λ_1 of the LCP of

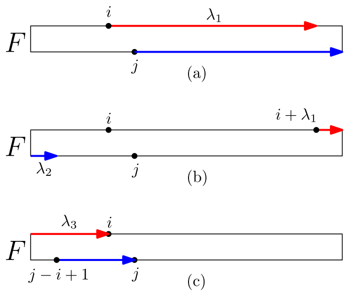


Fig. 2 Illustration of Lemma 7

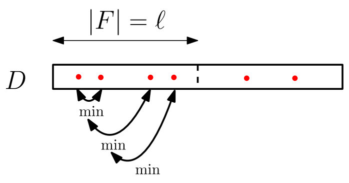


Fig. 3 Illustration of three applications of Lemma 7 on F. We mark the elements of A in red circles

$F[i\,..\,|F|]$ and $F[j\,..\,|F|]$. If $\lambda_1 < |F| - j + 1$, then we simply compare $F[i\,..\,i+\lambda_1]$ and $F[j\,..\,j+\lambda_1]$ to find the answer. If $\lambda_1 = |F| - j + 1$ (see top part (a)), we ask for the length λ_2 of the LCP of $F[i+\lambda_1\,..\,|F|]$ and F. If $\lambda_2 < j - i$, then we simply compare $F[i+\lambda_1\,..\,i+\lambda_1+\lambda_2]$ and $F[1\,..\,1+\lambda_2]$ to find the answer. If $\lambda_2 = j - i$ (see middle part (b)), we ask for the length λ_3 of the LCP of F and $F[j-i+1\,..\,|F|]$. If $\lambda_3 < j - i$, then we simply compare $F[1\,..\,1+\lambda_3]$ and $F[j-1\,..\,j-1+\lambda_3]$ to find the answer. Otherwise, rotation i of F is equal to rotation j of F (see bottom part (c)). □

We next use Lemma 7 as a building block to obtain Lemma 8, the main lemma used by our algorithm.

Lemma 8 *Let D be a string of length $|D| \geq \ell$. Let A be a set of d positions on D such that for every range $[i, i+\ell-1] \subseteq [1, |D|]$, $1 \leq i \leq |D| - \ell + 1$, there exists at least one element $j \in A \cap [i, i+\ell-1]$. Given a data structure for answering LCP$_D$ queries in $\mathcal{O}(1)$ time, we can find the smallest lexicographic rotation j in every length-ℓ fragment of D, such that $j \in A$ and j is minimal in $\mathcal{O}(d|D|)$ total time.*

Proof First we sort the elements of A in increasing order using radix sort in $\mathcal{O}(|D|)$ time. Then for every fragment $F = D[i\,..\,i+\ell-1]$ of length ℓ of D, $i \in [1, |D|-\ell+1]$, we consider pairs of elements from A that are in $[i, i+\ell-1]$. We can consider these pairs from left to right because the elements of A have been sorted and make $\mathcal{O}(d)$ comparisons for finding the minimum in a range. For any two elements, we perform an application of Lemma 7, which takes $\mathcal{O}(1)$ time, and maintain the leftmost smallest rotation. Inspect Figure 3.

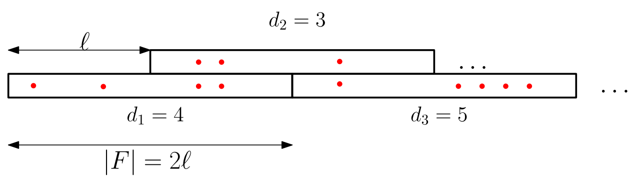


Fig. 4 Illustration of the decomposition of S. We mark the (w, k)-minimizers in red circles

Since we have at most $|D|$ length-ℓ fragments in D and at most $d-1$ pairs that we consider per fragment, the total time to process all fragments is $\mathcal{O}(d|D|)$. □

Theorem 2 *Given a string S of length n, randomly generated by a memoryless source over an alphabet Σ of size $\sigma \geq 2$ with identical letter probabilities, and an integer $\ell > 0$, the expected number of reduced bd-anchors of order ℓ for $r = \lceil 4 \log \ell / \log \sigma \rceil$ in S is $\mathcal{O}(n/\ell)$. Moreover, if Σ is an integer alphabet of size $\sigma = n^{\mathcal{O}(1)}$, $\mathcal{A}_{\ell,r}(S)$ can be computed in $\mathcal{O}(n)$ time on average.*

Proof The first part is merely Lemma 3 from [78].

For the second part, we employ the so-called standard trick to conceptually decompose S in q fragments F each of length $|F| = 2\ell$ overlapping by ℓ positions (apart perhaps from the last one). Inspect Figure 4. We compute $\mathcal{M}_{w,k}(S)$ for $w = \ell - r$ and $k = r + 1$. This can be done in $\mathcal{O}(n)$ time by Lemma 2 [77]. We also construct an LCP$_S$ data structure in $\mathcal{O}(n)$ time [61]. Let us denote by d_i the number of (w, k)-minimizers in the ith fragment F_i of S as per the above decomposition of S. We apply Lemma 8 with $D = F_i, d = d_i$ and $A = \mathcal{M}_{w,k}(F_i)$. This takes $d_i|F_i| = \mathcal{O}(d_i\ell)$ time. Note that $q = \Theta(n/\ell)$. Then the total work of the algorithm, by Lemma 8, is bounded *on average* by:

$$n + \ell d_1 + \ell d_2 + \ldots + \ell d_q = n + \ell(d_1 + d_2 + \ldots + d_q) \\ \leq n + 2\ell \cdot \mathcal{M}_{w,k}(S) = \mathcal{O}(n).$$

The last inequality holds by the fact that the fragments overlap by ℓ positions, which means that d_i will generally be considered twice (see Figure 4), and by Lemma 1, which gives an expected asymptotic upper bound on the number of (w, k)-minimizers in S. The algorithm is correct by Fact 1. □

In the worst case (e.g., $S = \mathtt{a}^n$), $\mathcal{M}_{w,k}(S) = \Theta(n)$, and so the algorithm of Theorem 2 takes $\Theta(n\ell)$ time.

Let us now highlight the differences for computing the set of randomized reduced bd-anchors. The following lemma follows directly from Lemma 2 assuming oracle access to a random order.

Lemma 9 *For any string S of length n over an integer alphabet and integers $w, k > 0$, the set $\mathcal{M}^h_{w,k}(S)$ can be computed in $\mathcal{O}(n)$ time assuming oracle access to a random order $h : \Sigma^k \to \mathbb{N}$.*

In practice, indeed, the KR fingerprints for all fixed-length fragments of S can be computed in $\mathcal{O}(n)$ time [59].

Theorem 3 follows directly from Theorem 2 by making the following two changes in the proof of Theorem 2:

- Replace Lemma 3 by Lemma 6.
- Replace Lemma 2 by Lemma 9.

Theorem 3 *Given a string S of length n, randomly generated by a memoryless source over an alphabet Σ of size $\sigma \geq 2$ with identical letter probabilities, and an integer $\ell > 0$, the expected number of randomized reduced bd-anchors of order ℓ for $r = \lceil 4 \log \ell / \log \sigma \rceil$ in S is $\mathcal{O}(n/\ell)$. Moreover, $\mathcal{A}^h_{\ell,r}(S)$ can be computed in $\mathcal{O}(n)$ time on average assuming oracle access to a uniformly random permutation h.*

5.2 Index construction in small space

In this section, we show the index construction in small space using near-optimal work. We show how to construct the index with reduced bd-anchors. For the randomized counterpart, maintaining a random order may take a lot of space [21]; in practice, we circumvent this problem using KR fingerprints.

Note that a *straightforward* implementation of the index $\mathcal{I}_{\ell,r}(S)$ (see also [77]) requires $\Theta(n)$ space in any case. In particular, this is because the SA and LCP array of S (and $\overleftarrow{S}$) are required for: **(i)** the implementation of Lemma 2 [77] (reduced bd-anchors); and **(ii)** the construction of $\mathcal{T}^R_{\ell,r}(S)$ and $\mathcal{T}^L_{\ell,r}(S)$ [77].

However, the size of $\mathcal{I}_{\ell,r}(S)$ can be asymptotically smaller than $\Theta(n)$. In fact, Theorem 1 tells us that the size of $\mathcal{I}_{\ell,r}(S)$ is $\mathcal{O}(|\mathcal{A}_{\ell,r}(S)|)$, which is expected to be $\mathcal{O}(n/\ell)$. Note that $|\mathcal{A}_{\ell,r}(S)| = \Omega(n/\ell)$ in any case [77], so we cannot hope for less memory. The goal of this section is to show that we can construct $\mathcal{I}_{\ell,r}(S)$ *efficiently* using only $\mathcal{O}(\ell + |\mathcal{A}_{\ell,r}(S)|)$ space (and external memory). (Note that in practical applications ℓ never exceeds n/ℓ and so ℓ is negligible.) We will show two constructions: one utilizing internal and external memory; and another one utilizing only internal memory.

In the standard *external memory* (EM) model [95], with internal memory (RAM) size M and disk block size B, the standard I/O complexities are: $\mathsf{scan}(n) = n/B$, which is the complexity of scanning n elements sequentially; and $\mathsf{sort}(n) = (n/B)\log_{M/B}(n/B)$, which is the complexity of sorting n elements [95]. *Semi-EM* model is a relaxation of the EM model, in which we are allowed to store some of the data in internal memory [1]. We proceed in four steps: In Step 1, we compute the set $\mathcal{A}_{\ell,r}(S)$ using $\mathcal{O}(\ell)$ extra space; in Step 2, we compute the SA and LCP array of S in the EM model using existing algorithms; in Step 3, we construct $\mathcal{T}^R_{\ell,r}(S)$ and $\mathcal{T}^L_{\ell,r}(S)$ in the semi-EM model (by having $\mathcal{A}_{\ell,r}(S)$ in internal memory); in Step 4, we construct the 2D range reporting data structure in internal memory using existing algorithms. The only difference of the second construction is that it skips Step 2, and directly constructs $\mathcal{T}^R_{\ell,r}(S)$ and $\mathcal{T}^L_{\ell,r}(S)$ utilizing only internal memory.

We next describe in detail how every step is implemented using near-optimal work.

Step 1 We make use of $\mathcal{O}(\ell)$ extra space. Specifically, we use Theorem 2 to construct $\mathcal{A}_{\ell,r}(S)$ via considering fragments of S of length-2ℓ (apart perhaps from the last one), overlapping by ℓ positions, without significantly increasing the time. If the size of the alphabet of a fragment F is not polynomial in ℓ (i.e., F consists of large integers), we use $\tilde{\mathcal{O}}(\ell)$ time instead of $\mathcal{O}(\ell)$ by first sorting the letters of F, and then assigning each letter to a rank in $\{1, \ldots, 2\ell\}$. This clearly does not affect the lexicographic rank of rotations. Thus the total time to construct $\mathcal{A}_{\ell,r}(S)$ is $\tilde{\mathcal{O}}(n)$: there are $\mathcal{O}(n/\ell)$ fragments and each one is processed in $\tilde{\mathcal{O}}(\ell)$ time. We finally implement $\mathcal{A}_{\ell,r}(S)$ as a perfect hash table [41], denoted by $\mathcal{H}_{\ell,r}(S)$, which we keep in RAM. This is done in $\mathcal{O}(|\mathcal{A}_{\ell,r}(S)|)$ time; and so the total time is $\tilde{\mathcal{O}}(n)$.

Step 2 We compute the SA and LCP array of S in external memory using M words of internal memory and disk block size B. To this end we can use existing algorithms, which compute the two arrays simultaneously [16, 71]; these algorithms are optimal with respect to internal work $\mathcal{O}(n \log_{M/B}(n/B))$ and I/O complexity $\mathcal{O}((n/B)\log_{M/B}(n/B))$—see also [66–70]. We also compute the SA and LCP array of $\overleftarrow{S}$, the reverse of S, analogously. For the internal memory construction, this step is omitted.

Step 3 We construct the two compacted tries $\mathcal{T}^R_{\ell,r}(S)$ and $\mathcal{T}^L_{\ell,r}(S)$ with the aid of four arrays, each of size $|\mathcal{A}_{\ell,r}(S)|$: RSA; RLCP; LSA; and LLCP. Specifically, RSA (Right SA) stores a permutation of $\mathcal{A}_{\ell,r}(S)$ such that $\mathsf{RSA}[i]$ is the starting position of the ith lexicographically smallest suffix of S with $\mathsf{RSA}[i] \in \mathcal{A}_{\ell,r}(S)$. $\mathsf{RLCP}[i]$ array (Right LCP array) stores the length of the LCP of $\mathsf{RSA}[i-1]$ and $\mathsf{RSA}[i]$. LSA and LLCP array are defined analogously for $\overleftarrow{S}$. Let us show how we construct RSA and RLCP; the other case for LSA and LLCP array is symmetric. To compute these arrays we use the SA and the LCP array of S constructed in Step 2. We scan the SA and the LCP array of S sequentially and sample them using the hash table $\mathcal{H}_{\ell,r}(S)$ constructed in Step 1. Let us suppose that we want to sample the kth value after reading $\mathsf{SA}[i]$ and $\mathsf{LCP}[i]$. This is possible using $\mathcal{O}(1)$ words of memory. If $\mathsf{SA}[i]$ is in $\mathcal{H}_{\ell,r}(S)$, which we check in $\mathcal{O}(1)$ time, we set $\mathsf{RSA}[k] = \mathsf{SA}[i]$. It is also well known that for any $i_1 < j_2$ the length of the LCP between $S[\mathsf{SA}[i_1] \mathinner{.\,.} n]$ and $S[\mathsf{SA}[i_2] \mathinner{.\,.} n]$ is the minimum value lying in $\mathsf{LCP}[i_1+1], \ldots, \mathsf{LCP}[i_2]$. Since we scan also the LCP array simultaneously, we maintain the value we need to store in $\mathsf{RLCP}[k]$. Finally we increment k and i by one. Scanning RSA and RLCP takes $\mathcal{O}(n/B)$ I/Os. Using RSA and RLCP we can construct $\mathcal{T}^R_{\ell,r}(S)$ in $\mathcal{O}(|\mathcal{A}_{\ell,r}(S)|)$ time using a folklore algorithm (cf. [60]). We construct $\mathcal{T}^L_{\ell,r}(S)$ analogously from LSA and LLCP. For the internal-memory construction, Step 3 can be implemented in $\tilde{\mathcal{O}}(n)$ time using $\mathcal{O}(|\mathcal{A}_{\ell,r}(S)|)$ space assuming read-only random access to S. This is achieved using any algorithm for *sparse suffix sorting* [5, 17, 64].

Step 4 By using RSA and LSA, we construct the 2D range reporting data structure in $\tilde{\mathcal{O}}(|\mathcal{A}_{\ell,r}(S)|)$ time using $\mathcal{O}(|\mathcal{A}_{\ell,r}(S)|)$ space [12, 23, 44, 85].

This completes the construction: we have shown how to construct $\mathcal{I}_{\ell,r}(S)$ in near-optimal work using only $\mathcal{O}(\ell + |\mathcal{A}_{\ell,r}(S)|)$ space (and external memory). This is good because the size of $\mathcal{I}_{\ell,r}(S)$ is $\mathcal{O}(|\mathcal{A}_{\ell,r}(S)|)$ (Theorem 1) and the $\mathcal{O}(\ell)$ term is negligible in practice.

6 Index with worst-case guarantees

The index based on bd-anchors has $\mathcal{O}(|\mathcal{A}_{\ell,r}(S)|)$ size, and supports pattern matching queries in $\tilde{\mathcal{O}}(|P| + \text{occ})$ time. Moreover, it can be constructed in $\tilde{\mathcal{O}}(n)$ time using $\mathcal{O}(\ell + |\mathcal{A}_{\ell,r}(S)|)$ space. Unfortunately, while in many real-world datasets we have $|\mathcal{A}_{\ell,r}(S)| = \Theta(n/\ell)$, there are worst cases with $|\mathcal{A}_{\ell,r}(S)| = \Theta(n)$; e.g., for $S = a_1a_2\dots a_n$, with $a_1 = a_2 = \dots = a_n$.

In this section, we combine some ideas presented in [8] with the bd-anchors index to prove the following.

Theorem 4 *For any string S of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$ and any integer $\ell > 0$, we can construct an index that occupies $\mathcal{O}(n/\ell)$ extra space and reports all occ occurrences of any pattern P of length $|P| \geq \ell$ in S in $\tilde{\mathcal{O}}(|P| + \text{occ})$ time. The index can be constructed in $\tilde{\mathcal{O}}(n)$ time and $\mathcal{O}(n/\ell)$ working space.*

We define one of the most basic string notions. An integer $p > 0$ is a *period* of a string P if $P[i] = P[i + p]$ for all $i \in [1, |P| - p]$. The smallest period of P is referred to as *the period* of P and is denoted by $\text{per}(P)$.

Example 6 For $P = \texttt{abaaabaaabaaaba}$, $\text{per}(P) = 4$; $p = 8$ is also a period of P but it is not the smallest.

We are now in a position to define the following powerful notion of locally consistent anchors.

Definition 4 (τ-partitioning set [64]) For any string S of length n and any integer $\tau \in [4, n/2]$, a set of positions $\mathcal{P} \subseteq [1, n]$ is called a *τ-partitioning set* if it satisfies the following properties:

1. if $S[i-\tau \mathinner{.\,.} i+\tau] = S[j-\tau \mathinner{.\,.} j+\tau]$, for $i, j \in [\tau, n-\tau)$, then $i \in \mathcal{P}$ if and only if $j \in \mathcal{P}$;
2. if $S[i \mathinner{.\,.} i + \ell] = S[j \mathinner{.\,.} j + \ell]$, for $i, j \in \mathcal{P}$ and some $\ell \geq 0$, then, for each $d \in [0, \ell - \tau)$, $i + d \in \mathcal{P}$ if and only if $j + d \in \mathcal{P}$;
3. if $i, j \in [1, n]$ with $j - i > \tau$ and $(i, j) \cap \mathcal{P} = \emptyset$, then $S[i \mathinner{.\,.} j]$ has period at most $\tau/4$.

Let us explain these properties. Property 1 tells us that for any two equal sufficiently long fragments of S, we either pick no anchor or we pick an anchor having the same relative position in both (local consistency). Property 2 tells us that for any two equal sufficiently long fragments of S, the anchors we pick from both are in sync (forward synchronization). Property 3 tells us that for a sufficiently long fragment of S, if no anchor is picked, then this fragment is highly periodic (approximately uniform sampling).

Theorem 5 *([64]) For any string S of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$ and any integer $\tau \in [4, \mathcal{O}(n/\log^2 n)]$, we can construct a τ-partitioning set $\mathcal{P}$ of size $\mathcal{O}(n/\tau)$. The set $\mathcal{P}$ can be constructed in $\tilde{\mathcal{O}}(n)$ time using $\mathcal{O}(n/\tau)$ working space. The set $\mathcal{P}$ additionally satisfies the property that if a fragment $S[i \mathinner{.\,.} j]$ has period at most $\tau/4$, then $\mathcal{P} \cap [i+\tau, j-\tau] = \emptyset$.*

We now define the notion of τ-runs: sufficiently long maximal fragments with the same periodic structure.

Definition 5 (τ-run [8]) A fragment F of a string S is a *τ-run* if and only if $|F| > 3\tau$, $\text{per}(F) \leq \tau/4$, and F cannot be extended in either direction without its period changing. The *Lyndon root* L of a τ-run F is the lexicographically smallest rotation of $L = F[1 \mathinner{.\,.} \text{per}(F)]$. The τ-run F with Lyndon root L has a canonical *representation* $L_1 \cdot L^{e_F} \cdot L_2$, where L_1 is a proper suffix of length x_F of L, L_2 is a proper prefix of length y_F of L, and $e_F \geq 0$ is a non-negative integer; either of strings L_1 and L_2 may be empty.

Example 7 For $F = \texttt{abaaabaaabaaaba}$ with $\text{per}(F) = 4$, the Lyndon root is $L = \texttt{aaab}$. Specifically, we have $x_F = 2$, $e_F = 3$, $y_F = 1$, and the representation:

$$F = \texttt{ab} \cdot (\texttt{aaab})^3 \cdot \texttt{a}.$$

We also state a few combinatorial and algorithmic results on τ-runs that are crucial in our construction.

Lemma 10 *([26]) Two τ-runs can overlap by at most $\tau/2$ positions. The number of τ-runs in a string of length n is $\mathcal{O}(n/\tau)$.*

Lemma 11 *([8]) For any string S of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$ and any integer $\tau \in [4, \mathcal{O}(n/\log^2 n)]$, all τ-runs in S can be computed and grouped by Lyndon root in $\tilde{\mathcal{O}}(n)$ time using $\mathcal{O}(n/\tau)$ space. Within the same complexities, we can compute the first occurrence of the Lyndon root in each τ-run.*

Data structure Let us start with an informal description of our data structure before presenting the details. We use Theorem 1 to construct an index over string S based on a set $\mathcal{A}$ of anchors. These anchors consist of the τ-partitioning set $\mathcal{P}$ and the set $\mathcal{L}$ of τ-runs of S, for some τ carefully selected as a function of ℓ. We further maintain $\mathcal{P}$ using KR fingerprints indexed using a hash table. We also maintain $\mathcal{L}$ using the KR fingerprints of the Lyndon roots and priority search trees. The latter trees are required to efficiently check whether a long periodic fragment of the pattern is included in a long periodic fragment of the text S.

Let ℓ be an integer in $[20, \lfloor n/\log^2 n\rfloor]$ and let $\tau = \lfloor \ell/5\rfloor$. We use Theorem 5 and Lemma 11 with parameter τ to compute a partitioning set $\mathcal{P}$ of size $\mathcal{O}(n/\tau)$ and all τ-runs in S. For every τ-run T of group L, we have the first occurrence of its Lyndon root. We can thus compute the non-negative integers $x_T, y_T < |L|$ and e_T, such that $T = L[|L|-x_T+1\mathinner{.\,.}|L|]\cdot L^{e_T}\cdot L[1\mathinner{.\,.}y_T]$, in constant time. We maintain the lists of pairs (e_T, x_T) and (e_T, y_T) for all T, sorted, for every L separately. This can be done in $\tilde{\mathcal{O}}(n)$ time and $\mathcal{O}(n/\tau)$ space using merge sort. To access the lists of L, we index the collection of lists by the KR fingerprint of L. We also maintain a priority search tree [81] over points (x_T, y_T), for every (L, e_T) separately, on the grid $[1, |L|]^2$. This can be done in $\tilde{\mathcal{O}}(n/\tau)$ time and $\mathcal{O}(n/\tau)$ space. To access a specific priority search tree for L, we further index the collection of trees by e_T. For every $i \in \mathcal{P}$ we also store the KR fingerprint (see Section 4) of $S[i-\tau\mathinner{.\,.}i+\tau]$ in a hash table H. This can be done in $\mathcal{O}(n)$ time using $\mathcal{O}(|\mathcal{P}|)$ space. Let $\mathcal{L}$ be a set that contains the starting and ending position of each τ-run. We construct $\mathcal{A} := \mathcal{P}\cup\mathcal{L}$ and construct the index of Section 3 over string S and $\mathcal{A}$. By Lemma 10 and Theorem 5 the size of $\mathcal{A}$ is $\mathcal{O}(n/\tau)$ in the worst case. By Theorem 5 and Theorem 11, $\mathcal{A}$ is constructed in $\tilde{\mathcal{O}}(n)$ time. By Theorem 1, the index over string S and $\mathcal{A}$ is constructed in $\tilde{\mathcal{O}}(n)$ time and occupies $\mathcal{O}(|\mathcal{A}|) = \mathcal{O}(n/\tau)$ extra space. The working space for the two compacted tries is also $\mathcal{O}(|\mathcal{A}|) = \mathcal{O}(n/\tau)$ by employing the following result on sparse suffix sorting.

Theorem 6 *([17, 64]) For any string S of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$, we can construct the compacted trie of b arbitrary suffixes of S in $\tilde{\mathcal{O}}(n)$ time using $\mathcal{O}(b)$ space.*

The working space for the accompanying 2D range reporting data structure is also $\mathcal{O}(|\mathcal{A}|) = \mathcal{O}(n/\tau)$ by using the data structure of Mäkinen and Navarro [85].

This completes the construction of the index which we denote by $\mathcal{I}_\ell(S)$. We have proved the following.

Lemma 12 *Index $\mathcal{I}_\ell(S)$, for any string S of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$ and any integer $\ell \in [20, \lfloor n/\log^2 n\rfloor]$, occupies $\mathcal{O}(n/\ell)$ extra space. Moreover, the index can be constructed in $\tilde{\mathcal{O}}(n)$ time and $\mathcal{O}(n/\ell)$ working space.*

Querying We are given a query P, $|P| \geq \ell$. Let us give an overview of the querying part: if P occurs in S, then it should be because it contains a sufficiently long aperiodic fragment, for which we have an anchor stored in $\mathcal{P} \subseteq \mathcal{A}$, or because it has a sufficiently long periodic fragment, which occurs also in S. The latter case is split in the following two subcases: (i) P has a periodic fragment shorter than P, in which case, we use one of its endpoints as an anchor stored in $\mathcal{L} \subseteq \mathcal{A}$; or the whole of P is periodic, in which case, we have no anchor to use, but we can use the Lyndon root of P to find its occurrences in S. We thus start by computing $\mathsf{per}(P)$ in $\mathcal{O}(|P|)$ time [30] and proceed as follows:

Aperiodic case: If $\mathsf{per}(P) > \tau/4$, we compute the KR fingerprint of every fragment $P[j-\tau\mathinner{.\,.}j+\tau]$ of P. If a KR fingerprint is found in H, we spell $P[j\mathinner{.\,.}|P|]$ and $\overleftarrow{P[1\mathinner{.\,.}j]}$ in the two compacted tries, and obtain the occurrences of P in S by using a 2D range reporting query just as in Section 3. The total query time is $\tilde{\mathcal{O}}(|P| + \mathsf{occ})$.

Fully-periodic case: In this case, $\mathsf{per}(P) \leq \tau/4$. We start by computing the Lyndon root L of P and the non-negative integers $x_P, y_P < |L|$ and e_P such that $P = L[|L|-x_P+1\mathinner{.\,.}|L|]\cdot L^{e_P}\cdot L[1\mathinner{.\,.}y_P]$ in $\mathcal{O}(|P|)$ time [8]. Let T be a τ-run in S with the same L. Further let $x_T, y_T < |L|$ and e_T such that $T = L[|L|-x_T+1\mathinner{.\,.}|L|]\cdot L^{e_T}\cdot L[1\mathinner{.\,.}y_T]$. P occurs in T if and only if at least one of the following conditions is met [8]: (1) $e_P = e_T$, $x_P \leq x_T$, and $y_P \leq y_T$; or (2) $e_P + 1 = e_T$ and $x_P \leq x_T$; or (3) $e_P + 1 = e_T$ and $y_P \leq y_T$; or (4) $e_P + 2 \leq e_T$. We first locate the sorted lists for the τ-runs T with Lyndon root L in S, in $\mathcal{O}(|L|)$ time, by computing the KR fingerprint of L, as well as the corresponding priority search trees. Each valid T is located in $\tilde{\mathcal{O}}(1)$ time with binary search on the stored lists or with one of the priority search trees. For instance, for checking the first condition, we locate the priority search tree for L and e_P and search for the points included in the axis-aligned rectangle $[x_P, \infty)\times[y_P, \infty)$. All s such points are found in $\mathcal{O}(\log n+s)$ time [81]. Let (x_T, y_T) be a retrieved point corresponding to τ-run T of S. This ensures that $e_P = e_T$, $x_P \leq x_T$, and $y_P \leq y_T$. The other three conditions are trivially checked with binary search. Since the length of T is $x_T + |L|\cdot e_T + y_T$; we know the length of P; and we also know the

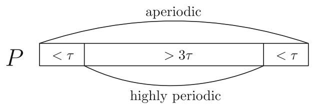


Fig. 5 The quasi-periodic case: in this setting, the long highly-periodic fragment is neither a prefix nor a suffix, and so we can use any of its endpoints as an anchor

first occurrence of L in both T and P, we can easily report all k occurrences of P in T in $\mathcal{O}(k)$ extra time. The total query time is $\tilde{\mathcal{O}}(|P| + \mathsf{occ})$.

Quasi-periodic case: It could be the case that P is not highly -periodic itself (i.e., $\mathsf{per}(P) > \tau/4$) but contains a long highly-periodic fragment that prevents us from finding a fragment $P[j-\tau \,..\, j+\tau]$ with a KR fingerprint in H (see Property 3, Definition 4). Inspect Figure 5. By the properties of $\mathcal{P}$ (Properties 1 and 3), we know that if P occurs in S then $\mathsf{per}(P[\tau \,..\, |P| - \tau + 1]) \le \tau/4$; note that, otherwise, we would have found at least one fragment of length $2\tau + 1$ whose KR fingerprint is in H.

Example 8 Let $\ell = 20$, $\tau = 4$, and $|P| = \ell$. Further assume that P occurs in S but $P[i - \tau \,..\, i + \tau]$, for all $i \in [5, 16]$ was not found in H. Property 3 of set $\mathcal{P}$ tells us that if P occurs in S then $P[4 \,..\, 17] = P[\tau \,..\, |P| - \tau + 1]$ has period at most $\tau/4$.

In $\mathcal{O}(|P|)$ time, we check if $\mathsf{per}(P[\tau \,..\, |P|-\tau+1]) \le \tau/4$ [30], and if so, we extend periodicity on both sides as much as possible, hence computing a periodic fragment $P[i \,..\, j]$, $i \le \tau$ and $j \ge |P|-\tau+1$. If $i > 1$, we construct $P[i \,..\, |P|]$ and $\overleftarrow{P}[1 \,..\, i]$; otherwise, if $j < |P|$, we construct $P[j \,..\, |P|]$ and $\overleftarrow{P}[1 \,..\, j]$. Note that it cannot be that both $i = 1$ *and* $j = |P|$, because then we would be in the previous case (fully-periodic case). We then spell the two sub-patterns in the two compacted tries, and obtain the occurrences of P in S by using a 2D range reporting query just as in Section 3. The total query time is $\tilde{\mathcal{O}}(|P| + \mathsf{occ})$.

Note that if any of the above three cases is successful in finding occurrences of P in S we stop, because we would in fact have found *all* the occurrences of P in S. On the other hand, if none of the above cases is successful, then it means that P does not occur in S.

This completes the querying algorithm on index $\mathcal{I}_\ell(S)$ for any P with $|P| \ge \ell$. We have proved the following.

Lemma 13 *Given index $\mathcal{I}_\ell(S)$, for any string S of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$ and any integer $\ell \in [20, \lfloor n/\log^2 n \rfloor]$, we can report all occ occurrences of any pattern P of length $|P| \ge \ell$ in S in $\tilde{\mathcal{O}}(|P| + \mathsf{occ})$ time.*

We are now ready to provide the proof of Theorem 4.

Proof of Theorem 4 When $\ell < 20 = \mathcal{O}(1)$, we simply construct the suffix tree of S (as our only index) in $\mathcal{O}(n)$ time [34]. The space occupied by the suffix tree is $\mathcal{O}(n)$, which for any $\ell = \mathcal{O}(1)$, is in $\mathcal{O}(n/\ell)$. When $\ell \in [20, \lfloor n/\log^2 n \rfloor]$, we apply Lemma 12 and then Lemma 13. When $\ell > \lfloor n/\log^2 n \rfloor$, then we do not construct any text index. Upon a query P, with $|P| > \lfloor n/\log^2 n \rfloor$, we use a real-time constant-space string matching algorithm [20] to report all the occurrences of P in S in $\mathcal{O}(n)$ time, in which case we have $\mathcal{O}(n) = \tilde{\mathcal{O}}(|P| + \mathsf{occ})$. □

Our construction is randomized (correct with high probability) due to the use of KR fingerprints.

7 Implementations

In this section, we provide the full details of our implementations, which have all been written in C++. In addition to the classic text indexes referred to in Section 1, we have considered the r-index [43], a text index, which performs specifically well for highly repetitive text collections. We did not consider: **(i)** the suffix tree, as it is not competitive at all with respect to space; or **(ii)** sampling the suffix array with minimizers [49], as their number is generally greater than (reduced) bd-anchors; see the results of [77, 78]. As we show later in experiments (Section 8), the set of randomized reduced bd-anchors can be computed *faster* and using *less space* than the set of reduced bd-anchors. Moreover, we show that the former set is *always smaller* than the latter. We have thus used randomized reduced bd-anchors instead of reduced bd-anchors to construct our indexes. We have also not implemented our index with worst-case guarantees (Section 6) as it relies on partitioning sets [64], an extremely intricate sampling scheme, which is highly unlikely to be efficient in practice,[5] as well as on the involved querying scheme consisting of many cases, which is also highly unlikely to be efficient in practice.

Since our focus is on algorithmic ideas (not on low-level code optimization) and to ensure fairness across different implementations, we have re-used the same algorithm or code whenever it was possible. For instance, although many other engineered implementations for classic text indexes exist, e.g. [46] for FM-index or [15] for suffix array, we have based

[5] Personal communication with Dmitry Kosolobov.

our implementations on libsais[6] and sdsl-lite [45] as much as possible for fairness:

SA: For suffix array construction of the smaller datasets, we used the libsais library, written by Ilya Grebnov. For the larger datasets, we have instead used Big-BWT [19] to infer the suffix array; we refer to it as SA (Big-BWT). To report all occurrences of a pattern, we implemented the algorithm of Manber and Myers [80] that uses as well the LCP array [60] augmented with a succinct RMQ data structure [39] (`rmq_succinct_sct` class).

CSA: The CSA was constructed using the `csa_sada` class of sdsl-lite. To report all occurrences of a pattern, we used the SA method.

CST: The CST was constructed using the `cst_sct3` class [89] of sdsl-lite. To report all occurrences of a pattern, we traverse the tree (see Section 2) by using its supported functionality.

FM-index: The FM-index was implemented using the `csa_wt` class of sdsl-lite. To report all occurrences of a pattern, the supported `count` and `locate` methods of sdsl-lite were used.

r-index: The r-index is not part of sdsl-lite and so we had to resort to the open-source implementation provided by Gagie et al. in [43]. The binary `ri-build` was used to build the index (using the `divsufsort` class of the sdsl-lite) and then `ri-locate` was used to locate all occurrences of the patterns.

rBDA-compute: We implemented the algorithm for computing the set of reduced bd-anchors as described in Section 5.1. To improve construction space, we implemented the algorithm in fragments as described in Step 1 of the space-efficient algorithm presented in Section 5.2. The implementation takes the fragment length b as input. We call these fragments *blocks* henceforth. To improve the runtime in practice, instead of implementing Lemma 7 using LCP queries, we have used naive letter comparisons [56].

rrBDA-compute: Similar to the above, we implemented the algorithm for computing the set of randomized reduced bd-anchors as described in Section 5.1. The implementation takes the block length b as input and uses naive letter comparisons instead of LCP queries. As shown in Section 8.2, randomized reduced bd-anchors perform consistently better than reduced bd-anchors. Hence, we have based our index construction implementations, presented next, on the notion of randomized reduced bd-anchors.

rMinimizers: We use the popular randomized minimizers scheme [100] where the ordering of length-k substrings is determined by the KR fingerprints and ties are broken by choosing the leftmost minimizer.

rrBDA-index I (ext): The construction was implemented using Steps 1 to 3 as described in Section 5.2. For Step 2, we used the pSAscan algorithm of Kärkkäinen et al. [69] to construct the SA in EM and the EM-SparsePhi algorithm of Kärkkäinen and Kempa [68] to compute the LCP array in EM. For Step 3, instead of constructing the compacted tries, we used the arrays LSA, LLCP, RSA, and RLCP directly. The LLCP and RLCP arrays were augmented each with an RMQ data structure (similar to SA). For Step 4, we implemented the 2D range reporting data structure of Mäkinen and Navarro [85]. To report all occurrences of a pattern, we implemented the algorithm of Loukides and Pissis [77] (see Section 3).

rrBDA-index I (int): For the internal-memory construction, we implemented Step 3 using the PA algorithm for sparse suffix sorting of Ayad et al. from [5]. Sparse suffix sorting is not included in sdsl-lite, and so we had to resort to PA, being the state of the art.

rrBDA-index II (ext): The construction was implemented using Steps 1 to 3 as in rrBDA-index I (ext). Unlike rrBDA-index I (ext), however, no 2D range reporting data structure was used. We instead used the bidirectional search algorithm presented by Loukides and Pissis in [77]. This algorithm reports all occurrences of a pattern P, by first searching for either $P[j \mathinner{.\,.} |P|]$ or $\overleftarrow{P}[1 \mathinner{.\,.} j]$ using the four arrays, where j is the reduced bd-anchor of $P[1 \mathinner{.\,.} \ell]$; and then using letter comparisons to verify the remaining part of candidate occurrences. This index was the most efficient one in practice in [77]; however, note that the query time is in $\Omega(n)$ in the worst case.

rrBDA-index II (int): Similar to rrBDA-index I, for the internal-memory construction, we implemented Step 3 using the PA algorithm of Ayad et al. from [5].

[6] https://github.com/IlyaGrebnov/libsais

Table 1 Datasets characteristics

Dataset	Length n	Alphabet size σ
DNA	209,714,087	15
PROTEINS	209,006,085	24
XML	204,198,133	95
SOURCES	166,552,125	225
ENGLISH	205,627,746	222
HUMAN	3,136,819,257	15
HAPLOTYPE	10,000,000,000	4
BIG-ENGLISH	4,850,679,290	95

For implementing KR fingerprints, we used a class written by Dominik Kempa.[7]

8 Experiments

8.1 Experimental setup

We used 8 real datasets; see Table 1 for their characteristics. The first 5 datasets (texts) are benchmark datasets that were downloaded from the Pizza&Chili corpus [35]. We also used 3 large datasets: (1) the *Homo sapiens* genome (version GRCh38.p14), which we denote by HUMAN; (2) a dataset produced from the 1000 Genomes Project,[8] which we denote by HAPLOTYPE; and (3) a dataset which is the concatenation of all 35,750 English text files from the Gutenberg Project without project related headers, which we denote by BIG-ENGLISH. The last two datasets were downloaded from https://github.com/koeppl/phoni and http://pages.di.unipi.it/rossano/big_english.gz, respectively. We generated patterns of length 32, 64, 128, 256, 512, and 1024 for the first 5 datasets with 500,000 distinct patterns created per length. The patterns were generated by selecting occurrences uniformly at random from the datasets. For the last 3 datasets, we generated patterns similarly but the length of the patterns was 64, 256, 1,024, 4,096, and 16,384.

The experiments ran on an Intel Xeon Gold 648 at 2.5GHz with 192GB RAM. All programs were compiled with `g++` version `10.2.0` at the `-O3` optimization level. Our code and datasets are freely available at https://github.com/lorrainea/rrBDA-index.

Parameters For rBDA-compute, rrBDA-compute, rrBDA-index I (int and ext) and rrBDA-index II (int and ext), we set ℓ to the pattern length and b to 25K, unless stated otherwise. By Lemma 6, a suitable value for r for random strings is $\lceil 4\log\ell/\log\sigma\rceil$. Tables 2 and 3 show that this is in fact the best choice for our datasets, and so we set $r = \lceil 4\log\ell/\log\sigma\rceil$. We also set M (RAM), used in rrBDA-index I (ext) and rrBDA-index II (ext), to $\Theta(|\mathcal{A}_{\ell,r}|)$ as this is the final size of index $\mathcal{I}_{\ell,r}(S)$.

Measures Four measures were used: index size; query time; construction space; and construction time; (see Section 1). To measure the query and construction time, the `steady_clock` class of `C++` was used with the elapsed time measured in nanoseconds (ns). To measure the index size for each implementation as accurately as possible, the data structures required for querying the patterns were output to a file in secondary memory. The text size was not counted as part of the index size for any implementation. The construction space was measured by recording the maximum resident set size in kilobytes (KB) using the `/usr/bin/time -v` command.

Results In Sections 8.2-8.7, we present the results for the datasets of Table 1. To avoid cluttering up the figures, we omit the results of rrBDA-index I (resp., int and ext), as they were consistently outperformed by rrBDA-index II (resp., int and ext) in all four measures used. In Section 8.8, we present a use case for mapping HiFi reads.

8.2 Comparing reduced bd-anchors to randomized reduced bd-anchors

In the following, we compare our new notion of randomized reduced bd-anchors (rrBDA-compute) to reduced bd-anchors (rBDA-compute) along three dimensions (density, construction time, and construction space) using the first five datasets of Table 1. We also compare it to randomized minimizers (rMinimizers) in terms of density as the golden standard for density.

Figure 6 shows that the number of reduced bd-anchors (rBDA-compute) is larger than that of randomized reduced bd-anchors (rrBDA-compute) for all tested ℓ values and across all datasets. It also shows that the number of randomized minimizers (rMinimizers) is essentially the same to the latter. The normalization of those numbers by $\frac{n}{\ell}$ shows that the constant for all schemes is generally small (between 1.8 and 5.1).

Figure 7 shows that it is much more efficient to compute our notion of randomized reduced bd-anchors compared to reduced bd-anchors, and this is consistent over all tested ℓ values and datasets. Specifically, the time needed by rrBDA-compute was smaller than that of rBDA-compute by 32.4% on average (over all datasets) and up to 77.6% lower. On the other hand, the simple $\Theta(n\ell)$-time algorithm of [77], which computes reduced bd-anchors and is represented by the $\Theta(n\ell)$ line, is even slower than rBDA-compute. In particular, rBDA-compute becomes more than *two orders of magnitude faster* than the $\Theta(n\ell)$-time algorithm for $\ell > 2^5$.

[7] https://github.com/dominikkempa/lz77-to-slp

[8] This dataset was produced from the prefix of 10^{10} letters of the raw sequence which contains only `A`, `C`, `G`, and `T`.

Table 2 Number of randomized reduced bd-anchors for varying r values and $\ell = 2^8$ for the benchmark datasets

r	$\lceil 3\log \ell/\log \sigma \rceil$	$\lceil 4\log \ell/\log \sigma \rceil$	$\lceil 5\log \ell/\log \sigma \rceil$	$\lceil 6\log \ell/\log \sigma \rceil$	$\ell/2$
DNA	1685630	1692652	1703569	1717243	3225764
PROTEINS	1663205	1661459	1673162	1688785	3214744
XML	1683452	1422135	1718535	1807198	3143646
SOURCES	1598048	1604154	1620757	1632470	2635256
ENGLISH	1627553	1638049	1636320	1640331	3163281
Average	1651577.6	1603689.8	1670468.6	1697205.4	3076538.2

Table 3 Number of randomized reduced bd-anchors for varying r values and $\ell = 2^{10}$ for the large datasets

r	$\lceil 3\log \ell/\log \sigma \rceil$	$\lceil 4\log \ell/\log \sigma \rceil$	$\lceil 5\log \ell/\log \sigma \rceil$	$\lceil 6\log \ell/\log \sigma \rceil$	$\ell/2$
HUMAN	6218839	6206176	6236596	6235489	12206012
HAPLOTYPE	19850773	19916381	20054138	20127868	38948496
BIG-ENGLISH	9852505	9699883	9738123	9800940	18879038
Average	11974039	11940813.3	12009619	12054765.7	23344515.3

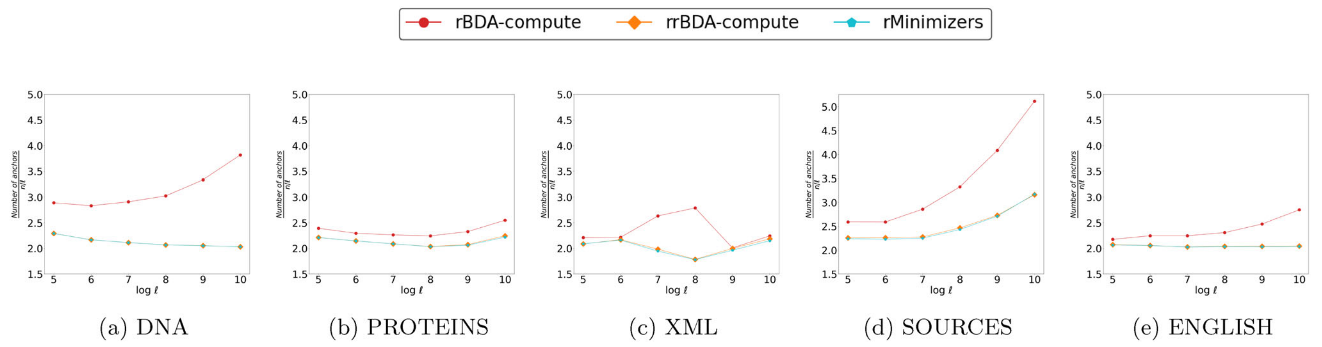


Fig. 6 Number of bd-anchors output by rBDA-compute and rrBDA-compute as well as the number of randomized minimizers output by rMinimizers for varying ℓ (log-scale). The numbers are normalized by n/ℓ

As suggested by Theorems 2 and 3, the runtime of rBDA-compute and of rrBDA-compute should not be affected by ℓ in the case of a uniformly random input string. Even if our datasets are real and thus not uniformly random, we observe that the runtime of rBDA-compute and rrBDA-compute in Figure 7 is not affected too much by the value of ℓ; an exception to this is the case of SOURCES, which is explained by the fact that this dataset, in particular, is very far from being *uniformly random*, which is assumed by our theoretical findings. Specifically, in this dataset, the number of (w, k)-minimizers is much larger than what is expected on average with increasing ℓ, thus more ties need to be broken (Lemma 8), and this is why the runtime increases with ℓ. Nevertheless, even in this bad case, rBDA-compute is up to 112× faster than the implementation of the simple $\Theta(n\ell)$-time algorithm [77], and rrBDA-compute is up to 177× faster.

Figure 8 shows that, for all tested ℓ values and datasets, rBDA-compute and rrBDA-compute have a *similar* trend to the $\Theta(n\ell)$-time implementation [77]. In particular, as ℓ increases the space required to construct bd-anchors is reduced. This is expected since for increasing ℓ the size of the set of bd-anchors decreases. Also, note that the space for rrBDA-compute is smaller by 12.8% on average (over all datasets) compared to that for rBDA-compute and by up to 49.1% compared to that for rBDA-compute. These space-time results establish the practical usefulness of our Theorems 2 and 3.

8.3 Index size

Figure 9 shows the index size for the first five datasets of Table 1 when using the rrBDA-compute implementation of Section 7 and rrBDA-index II (both `int` and `ext` versions have the same index size). As expected, the index size occupied by rrBDA-index II decreases with increasing ℓ. Notably, for all datasets and $\ell \geq 2^6$, rrBDA-index II is smaller than all other indexes except for the FM-index. This is remarkable, given that our index is not compressed and as we will show it allows much faster querying. When $\ell \geq 2^9$, rrBDA-index II *outperforms all indexes* for all datasets. Specifically, when

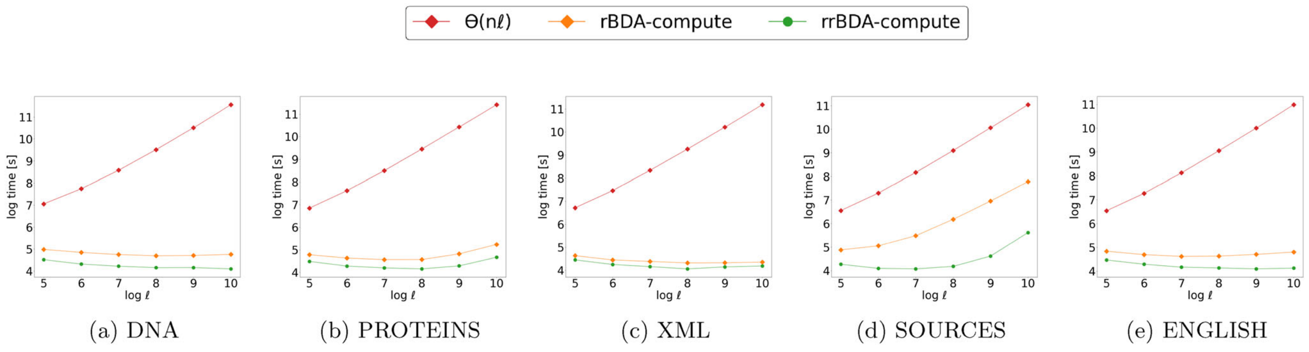


Fig. 7 Elapsed time to construct the set of reduced bd-anchors and randomized reduced bd-anchors (seconds in log-scale) for varying ℓ (log-scale)

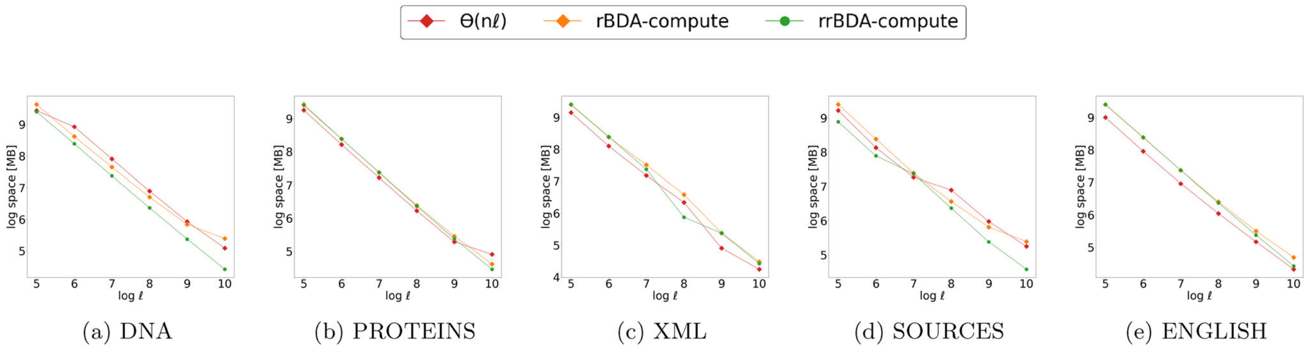


Fig. 8 Space required to construct the set of reduced bd-anchors and randomized reduced bd-anchors (MB in log-scale) for varying ℓ (log-scale)

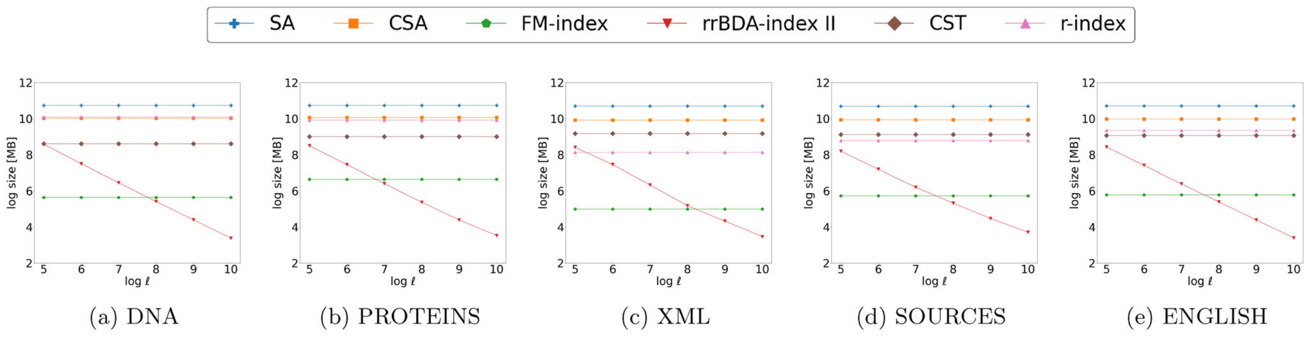


Fig. 9 Size of different indexes (MB in log-scale) for varying ℓ (log-scale)

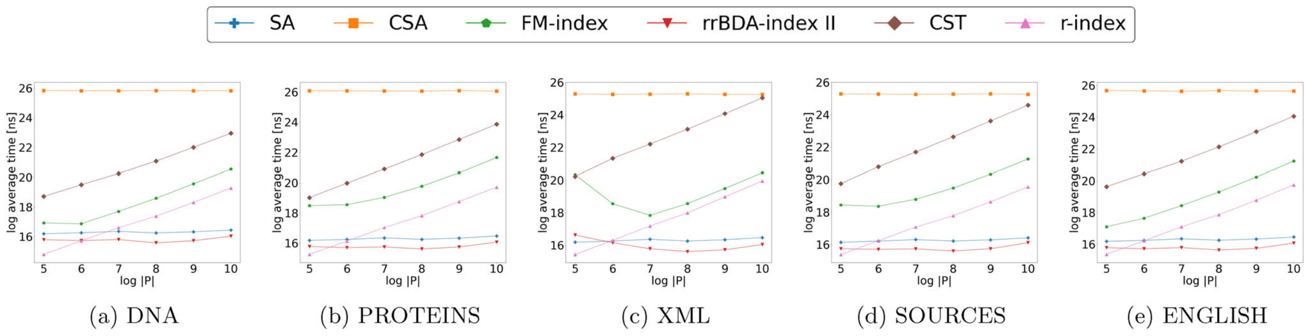


Fig. 10 Average time for pattern matching (nanoseconds in log-scale) for varying $|P|$ (log-scale)

$\ell = 2^9$, rrBDA-index II requires 59.1% less space than FM-index on average (over all datasets), while when $\ell = 2^{10}$ our index requires 77.9% less space.

8.4 Query time

Figure 10 shows the average query time (over all patterns) for the first five datasets in Table 1 when $\ell = |P|$. Both rrBDA-index II (int) and rrBDA-index II (ext) have the same query time, so we write rrBDA-index II to refer to either. For all datasets and $\ell \geq 2^6$, rrBDA-index II is up to *several orders of magnitude* faster than the compressed indexes, especially for large alphabets, which is consistent with the observations made in [37, 47]. Notably, rrBDA-index II is even faster than the SA, with the single exception of $\ell = 2^5$ in the XML dataset.

8.5 Index construction space

Figure 11 shows the index construction space required for the first five datasets of Table 1. Notably, for all datasets when $\ell \geq 2^7$, rrBDA-index II (int) and rrBDA-index II (ext) outperform all other indexes. As expected, the construction space required by rrBDA-index II (int) decreases with increasing ℓ. The construction space of rrBDA-index II (ext) decreases with ℓ initially but remains almost the same for $\ell \geq 2^6$. This is because approximately 600MB of minimum space (internal memory) are used by the implementations constructing arrays SA and LCP in external memory.

8.6 Index construction time

Figure 12 shows the time needed to construct all indexes for the first five datasets of Table 1. We set $b = 25$K for rrBDA-index II (int) and rrBDA-index II (ext). For all datasets and ℓ values, rrBDA-index II (ext) outperforms CSA and is outperformed by the SA, FM-index, CST, and r-index. It also outperforms rrBDA-index II (int) for $\ell = 2^5$ but performs worse for all larger ℓ values. This is because, as ℓ increases, the number of bd-anchors decreases significantly, and thus rrBDA-index II (int) becomes faster (e.g., it performs similarly to one or more of the competitors for $\ell = 2^{10}$). On the other hand, as ℓ increases, rrBDA-index II (ext) still needs to construct arrays SA and LCP in external memory, which is the bottleneck for the construction time.

8.7 Results for large datasets

The results in Figure 13a are analogous to the ones in Figure 6. Figure 13b shows that rrBDA-index II (both int and ext versions have the same index size) outperforms FM-index, r-index, and SA (Big-BWT) for $\ell \geq 2^6$ in terms of index size (we excluded the other indexes as they were worse). For example, for $\ell = 2^{14}$, rrBDA-index II takes less than 16MB, while FM-index, r-index, and SA (Big-BWT) take 1GB, 16GB, and 48GB, respectively.

Figure 13c shows the average query time for HUMAN with $\ell = |P|$. As ℓ increases, rrBDA-index II becomes more than one order of magnitude faster than the FM-index and r-index. Note that for all ℓ, rrBDA-index II is faster than the FM-index and r-index. rrBDA-index II is also faster than SA (Big-BWT) except when $\ell = 2^{14}$. Further note that, for $|P| \geq 1024$, the average query time of rrBDA-index II starts growing because the $\mathcal{O}(|P|)$ term in the time complexity of the querying algorithm starts to become more and more significant.

Figure 13d shows that rrBDA-index II (int) outperforms all other indexes in terms of construction space, and the difference between rrBDA-index II (int) and these indexes becomes larger as ℓ increases. The results are analogous to those in Figure 11. For example, for $\ell = 2^{14}$ rrBDA-index II (int) requires at least $2\times$ less space than rrBDA-index II (ext) and more than $4\times$ less space than FM-index and SA (Big-BWT). At the same time, rrBDA-index II (ext) substantially outperforms FM-index, r-index, and SA (Big-BWT) for all tested ℓ values.

Figure 13e shows that rrBDA-index II (int) performs better as ℓ increases, and even outperforms the best competitor, FM-index for $\ell > 2^{10}$.[9] At the same time, rrBDA-index II (ext) performs worse than FM-index and r-index, but better than SA (Big-BWT). The reason that rrBDA-index II (ext) performs worse than FM-index and r-index is the construction of SA and the LCP array, as explained before. The results are analogous to those for DNA in Figure 12.

The results in Figure 14a are analogous to the ones in Figure 13a. Figure 14b shows a similar result to that of Figure 13b but in HAPLOTYPE r-index performs better compared to HUMAN, as the former dataset is repetitive (i.e., the type of data that r-index was designed for). Yet, for $\ell \geq 2^{12}$ rrBDA-index II outperforms r-index, which is encouraging.

Figure 14c and 14d shows a similar result to that of Figure 13c and 13d, respectively but again r-index performs better in HAPLOTYPE compared to HUMAN for the same reason. However, rrBDA-index II (int) outperforms r-index for all tested ℓ values in terms of construction space and all but the smallest tested ℓ value in terms of query time. Figure 14d shows a similar result to that of 13d; r-index performs better compared to HUMAN but is still worse than both rrBDA-index II versions. Figure 14e shows a similar result to that of Figure 13e but r-index performs better as HAPLOTYPE is repetitive.

Figure 15a, 15b, and 15c shows a similar result to that of Figure 13a, 13b, and 13c, respectively. Figure 15d shows

[9] We set $b = 130K$, as this dataset is much larger than those above [4].

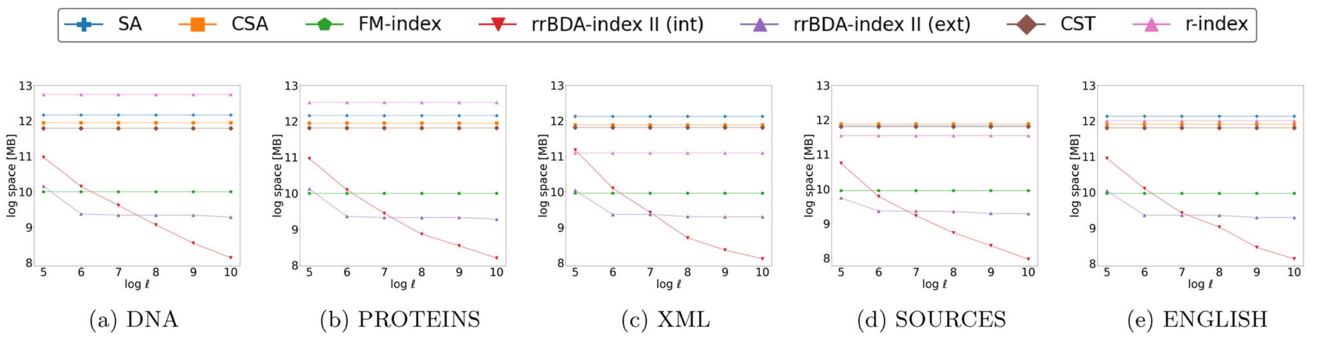


Fig. 11 Space required to construct different indexes (MB in log-scale) for varying ℓ (log-scale)

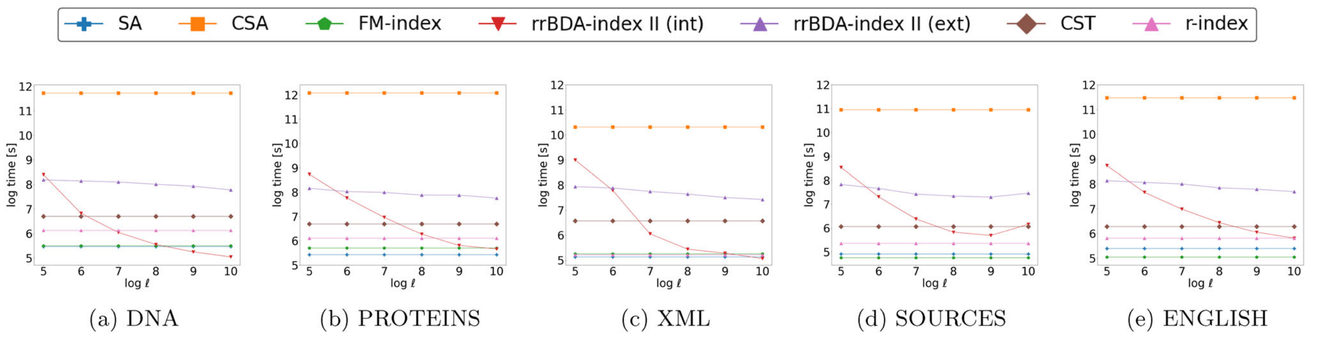


Fig. 12 Elapsed time to construct different indexes (seconds in log-scale) for varying ℓ (log-scale)

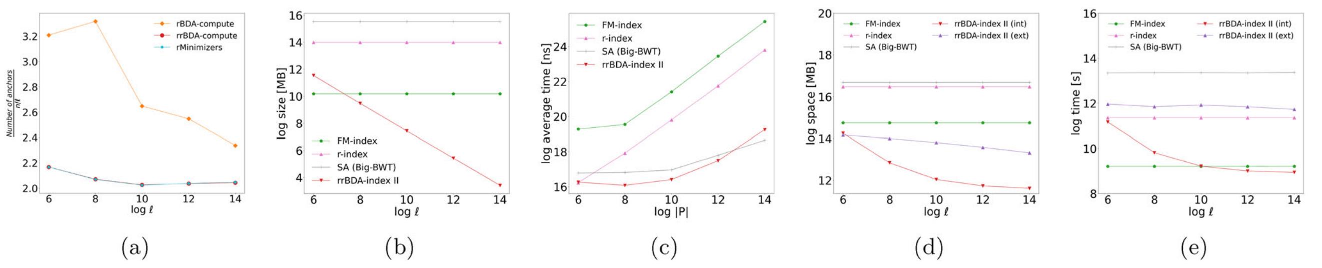


Fig. 13 (a) Number of bd-anchors output by rBDA-compute and rrBDA-compute as well as the number of randomized minimizers output by rMinimizers for HUMAN and varying ℓ (log-scale). The numbers are normalized by n/ℓ. (b) Size of different indexes (MB in log-scale) for HUMAN and varying ℓ (log-scale). (c) Average time for pattern matching (nanoseconds in log-scale) on HUMAN and for varying $|P|$ (log-scale). (d) Space required to construct different indexes (MB in log-scale) for HUMAN and varying ℓ (log-scale). (e) Elapsed time to construct different indexes (seconds in log-scale) for HUMAN and varying ℓ (log-scale)

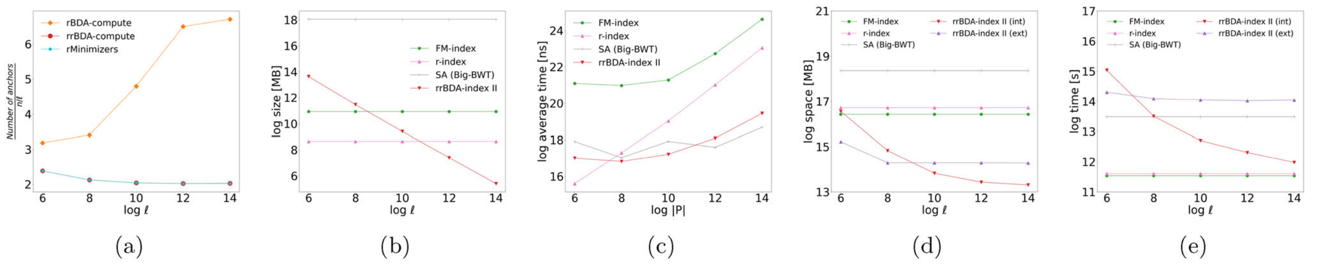


Fig. 14 (a) Number of bd-anchors output by rBDA-compute and rrBDA-compute as well as the number of randomized minimizers output by rMinimizers for HAPLOTYPE and varying ℓ (log-scale). The numbers are normalized by n/ℓ. (b) Size of different indexes (MB in log-scale) for HAPLOTYPE and varying ℓ (log-scale). (c) Average time for pattern matching (nanoseconds in log-scale) on HAPLOTYPE and for varying $|P|$ (log-scale). (d) Space required to construct different indexes (MB in log-scale) for HAPLOTYPE and varying ℓ (log-scale). (e) Elapsed time to construct different indexes (seconds in log-scale) for HAPLOTYPE and varying ℓ (log-scale)

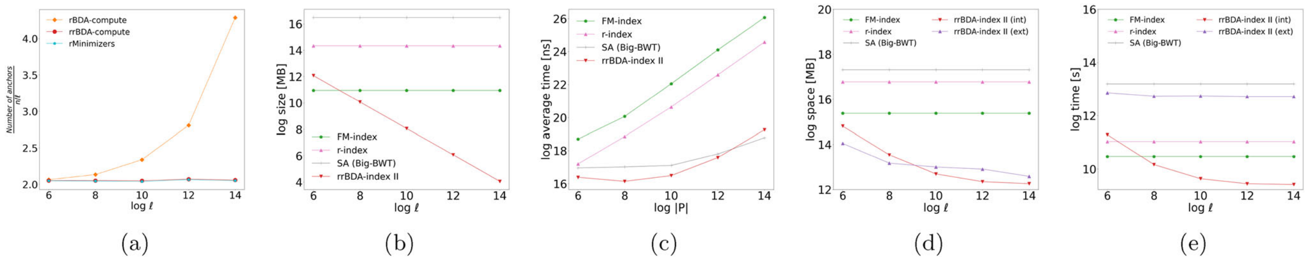


Fig. 15 (a) Number of bd-anchors output by rBDA-compute and rrBDA-compute as well as the number of randomized minimizers output by rMinimizers for BIG-ENGLISH and varying ℓ (log-scale). The numbers are normalized by n/ℓ. (b) Size of different indexes (MB in log-scale) for BIG-ENGLISH and varying ℓ (log-scale). (c) Average time for pattern matching (nanoseconds in log-scale) on BIG-ENGLISH and for varying $|P|$ (log-scale). (d) Space required to construct different indexes (MB in log-scale) for BIG-ENGLISH and varying ℓ (log-scale). (e) Elapsed time to construct different indexes (seconds in log-scale) for BIG-ENGLISH and varying ℓ (log-scale)

that both versions of rrBDA outperform all other indexes, as is the case for the HUMAN dataset. Figure 15e shows that rrBDA-index II (int) outperforms all indexes for $\ell \geq 2^8$; the performance of all these indexes is analogous to that in Figure 13e. The reason for this is that BIG-ENGLISH has a much larger alphabet than HUMAN and hence all suffixes share shorter LCPs. In this case, the PA algorithm used in rrBDA-index II (int) takes linear time; see Theorem 2 in [5].

The presented results highlight the scalability of our construction algorithms as well as the superiority of rrBDA-index II for long patterns.

8.8 Use case: Mapping PacBio long reads

To show that our index can facilitate approximate matching, an experiment was carried out using real highly accurate PacBio long read sequences, specifically using the HG002-rep1 dataset: 450 reads from this dataset were used.[10] These reads are approximately 99.9% accurate and have an average length of ∼16,000bp. Each read was decomposed into a collection of patterns, each of length 150. rrBDA-index was used with $\ell = 150$ to locate the patterns in the human genome (v. GRCh38.p14). The index size was 1.23GB and the average pattern matching time was 19ms. Indeed, *at least one pattern* in 439 reads out of 450 was located (matched) in the genome. These matches can be used as anchors (seeds) to map the reads fully onto the genome using dynamic programming. The same computation was done using the FM-index, which is commonly used in bioinformatics: the index size was 1.17GB but the average pattern matching time was 242ms, which is an order of magnitude larger than that using the rrBDA-index.

[10] Downloaded from https://downloads.pacbcloud.com/public/revio/2022Q4/HG002-rep1/.

9 Final remarks and future work

We have shown using 8 real datasets from different domains that our implementation of the bd-anchors index should be the practitioners' choice for long patterns, offering remarkably good performance with respect to all four measures and most of the times even outperforming the best index with respect to a certain measure. The construction time of our index is up to 5× slower than the state of the art. However, we believe that this does not make our index less practical as in many cases of practical interest our index can be constructed only once and queried multiple times.

An obvious criticism against the bd-anchors index is that its theoretical guarantees rely on the average-case model and on the assumption that we have at hand a lower bound on ℓ. For the former, we have shown, using benchmark datasets (and not using synthetic uniformly random ones), that our index generally outperforms the state of the art for long patterns, which is the main claim here. For the latter, in most real-world applications we can think of, we do have a lower bound on the length of patterns (see Section 1.2). For example, in long-read alignment, this bound is in the order of several hundreds. Finally, the state-of-the-art compressed indexes also have a crucial assumption, which is implicit and thus oftentimes largely neglected. The assumption is that the size σ of the alphabet is much smaller than the length n of the text. In the worst case, i.e., when $\sigma = \Theta(n)$, the state-of-the-art compressed indexes have no advantage compared to the suffix array or the suffix tree: they occupy $n \log \sigma = \Theta(n \log n)$ bits.

The theoretical highlight of this paper is Theorem 4. The underlying index has $\mathcal{O}(n/\ell)$ size, it can be constructed in near-optimal time using $\mathcal{O}(n/\ell)$ working space, and it supports near-optimal pattern matching queries. Since it works for any integer $\ell \in [1, n]$, it can be seen as a parameterization of the classic suffix tree index [34]. Our construction is currently randomized; it is an interesting open problem to make it deterministic.

Our immediate next goal is to investigate the possibility of improving the *construction time* of our practical index, while maintaining the excellent performance in the other three measures. To this end, in particular, we would like to explore the following two directions:

– A more exhaustive comparison between different lc-anchor schemes should be made to have the best trade-off between the index size and the construction time. It could even be the case that a hybrid approach would be the most desirable solution: e.g., use some type of minimizers for the aperiodic parts of the text; and then use a more sophisticated sampling scheme for the periodic parts of the text.
– A large part of the construction time is currently spent on each of the two "directions" of our bidirectional index. One could think of ways to appropriately amend the sampling scheme so that one of the two directions becomes irrelevant. For instance, using $r = \ell/2$ results in more bd-anchors, but then the anchor is always within the first half of $P[1\,..\,\ell]$, and so LSA and LLCP is practically irrelevant.

Note that KR fingerprints do not induce a uniformly random order on the fixed-length substrings of S; it remains open to prove Lemma 5 for KR fingerprints. An interesting question is whether the assumption about oracle access to h can be dropped in Theorem 3.

Another direction could be showing a stronger variant of Theorem 3, where the $\mathcal{O}(n/\ell)$ bound holds not only for random strings but, in general, for all strings without highly-periodic substrings. Finally, an interesting problem is whether the naive LCP queries employed by our implementation suffice to achieve the $\tilde{\mathcal{O}}(n)$ construction time for random strings (see Theorem 1).

Funding This work is partially supported by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively. The authors thank Panagiotis Charalampopoulos for helpful suggestions on Theorem 4.

References

1. Abello, J., Buchsbaum, A.L., Westbrook, J.R.: A functional approach to external graph algorithms. Algorithmica **32**(3), 437–458 (2002)
2. Apostolico, A., Crochemore, M., Farach-Colton, M., Galil, Z., Muthukrishnan, S.: 40 years of suffix trees. Commun. ACM **59**(4), 66–73 (2016)
3. Ariannezhad, M., Montazeralghaem, A., Zamani, H., Shakery, A.: Improving retrieval performance for verbose queries via axiomatic analysis of term discrimination heuristic. In: Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, Shinjuku, Tokyo, Japan, August 7-11, 2017, pp. 1201–1204. ACM (2017)
4. Ayad, L.A.K., Loukides, G., Pissis, S.P.: Text indexing for long patterns: anchors are all you need. Proc. VLDB Endow. **16**(9), 2117–2131 (2023)
5. Ayad, L.A.K., Loukides, G., Pissis, S.P., Verbeek, H.: Sparse suffix and LCP array: simple, direct, small, and fast. In: LATIN 2024: Theoretical Informatics - 16th Latin American Symposium, Puerto Varas, Chile, March 18-22, 2024, Proceedings, Part I, Lecture Notes in Computer Science, vol. 14578, pp. 162–177. Springer (2024)
6. Baeza-Yates, R.A., Perleberg, C.H.: Fast and practical approximate string matching. Inf. Process. Lett. **59**(1), 21–27 (1996)
7. Barbay, J., Claude, F., Gagie, T., Navarro, G., Nekrich, Y.: Efficient fully-compressed sequence representations. Algorithmica **69**(1), 232–268 (2014)
8. Bathie, G., Charalampopoulos, P., Starikovskaya, T.: Internal pattern matching in small space and applications. In: 35th Annual Symposium on Combinatorial Pattern Matching, CPM 2024, June 25-27, 2024, Fukuoka, Japan, LIPIcs, pp. 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024)
9. Belazzougui, D.: Linear time construction of compressed text indices in compact space. In: Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014, pp. 148–193. ACM (2014)
10. Belazzougui, D., Cunial, F., Kärkkäinen, J., Mäkinen, V.: Linear-time string indexing and analysis in small space. ACM Trans. Algorithms **16**(2), 17:1-17:54 (2020)
11. Belazzougui, D., Navarro, G.: Optimal lower and upper bounds for representing sequences. ACM Trans. Algorithms **11**(4), 31:1-31:21 (2015)
12. Belazzougui, D., Puglisi, S.J.: Range predecessor and lempel-ziv parsing. In: Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016, pp. 2053–2071. SIAM (2016)
13. Ben-Nun, S., Golan, S., Kociumaka, T., Kraus, M.: Time-space tradeoffs for finding a long common substring. In: 31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark, LIPIcs, vol. 161, pp. 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
14. Bendersky, M., Croft, W.B.: Discovering key concepts in verbose queries. In: Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2008, Singapore, July 20-24, 2008, pp. 491–498. ACM (2008)
15. Bertram, N., Ellert, J., Fischer, J.: Lyndon words accelerate suffix sorting. In: 29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference), LIPIcs, pp. 15:1–15:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)

16. Bingmann, T., Fischer, J., Osipov, V.: Inducing suffix and LCP arrays in external memory. ACM J. Exp. Algorithmics **21**(1), 2.3:1-2.3:27 (2016)
17. Birenzwige, O., Golan, S., Porat, E.: Locally consistent parsing for text indexing in small space. In: Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020, pp. 607–626. SIAM (2020)
18. Boncz, P.A., Neumann, T., Leis, V.: Fsst: fast random access string compression. Proc. VLDB Endow. **13**(11), 2649–2661 (2020)
19. Boucher, C., Gagie, T., Kuhnle, A., Langmead, B., Manzini, G., Mun, T.: Prefix-free parsing for building big BWTs. Algorithms Mol. Biol. **14**(1), 13:1-13:15 (2019)
20. Breslauer, D., Grossi, R., Mignosi, F.: Simple real-time constant-space string matching. Theor. Comput. Sci. **483**, 2–9 (2013)
21. Broder, A.Z., Charikar, M., Frieze, A.M., Mitzenmacher, M.: Min-wise independent permutations (extended abstract). In: Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998, pp. 327–336. ACM (1998)
22. Burkhardt, S., Kärkkäinen, J.: Fast lightweight suffix array construction and checking. In: Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings, Lecture Notes in Computer Science, vol. 2676, pp. 55–69. Springer (2003)
23. Chan, T.M., Larsen, K.G., Patrascu, M.: Orthogonal range searching on the RAM, revisited. In: Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011, pp. 1–10. ACM (2011)
24. Charalampopoulos, P., Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Pissis, S.P., Radoszewski, J., Rytter, W., Walen, T.: Linear-time algorithm for long LCF with k mismatches. In: Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China, *LIPIcs*, vol. 105, pp. 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
25. Charalampopoulos, P., Kociumaka, T., Pissis, S.P., Radoszewski, J.: Faster algorithms for longest common substring. In: 29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference), LIPIcs, pp. 30:1–30:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
26. Charalampopoulos, P., Kociumaka, T., Pissis, S.P., Radoszewski, J.: Faster algorithms for longest common substring. CoRR arXiv:abs/2105.03106 (2021)
27. Charalampopoulos, P., Pissis, S.P., Radoszewski, J.: Longest palindromic substring in sublinear time. In: 33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic, LIPIcs, vol. 223, pp. 20:1–20:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
28. Claude, F., Navarro, G., Peltola, H., Salmela, L., Tarhio, J.: String matching with alphabet sampling. J. Discrete Algorithms **11**, 37–50 (2012)
29. Cole, R., Kopelowitz, T., Lewenstein, M.: Suffix trays and suffix trists: structures for faster text indexing. Algorithmica **72**(2), 450–466 (2015)
30. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, United Kingdom (2007)
31. Dietzfelbinger, M., Gil, J., Matias, Y., Pippenger, N.: Polynomial hash functions are reliable (extended abstract). In: Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings, pp. 235–246. Springer (1992)
32. Dinklage, P., Fischer, J., Herlez, A.: Engineering predecessor data structures for dynamic integer sets. In: 19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France, LIPIcs, vol. 190, pp. 7:1–7:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
33. Dinklage, P., Fischer, J., Herlez, A., Kociumaka, T., Kurpicz, F.: Practical performance of space efficient data structures for longest common extensions. In: 28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference), pp. 39:1–39:20 (2020)
34. Farach, M.: Optimal suffix tree construction with large alphabets. In: 38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997, pp. 137–143 (1997)
35. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: from theory to practice. ACM J. Exp. Algorithmics **13**, (2008)
36. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005)
37. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: An alphabet-friendly FM-index. In: String Processing and Information Retrieval, 11th International Conference, SPIRE 2004, Padova, Italy, October 5-8, 2004, Proceedings, Lecture Notes in Computer Science, vol. 3246, pp. 150–160. Springer (2004)
38. Fischer, J., Gawrychowski, P.: Alphabet-dependent string searching with wexponential search trees. In: Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings, Lecture Notes in Computer Science, pp. 160–171. Springer (2015)
39. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. **40**(2), 465–492 (2011)
40. Franceschini, G., Muthukrishnan, S.: In-place suffix sorting. In: Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings, Lecture Notes in Computer Science, vol. 4596, pp. 533–545. Springer (2007)
41. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with 0(1) worst case access time. J. ACM **31**(3), 538–544 (1984)
42. Fredman, M.L., Willard, D.E.: Blasting through the information theoretic barrier with fusion trees. In: Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA, pp. 1–7. ACM (1990)
43. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in BWT-runs bounded space. J. ACM **67**(1), 2:1-2:54 (2020)
44. Gao, Y., He, M., Nekrich, Y.: Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing. In: 28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference), pp. 54:1–54:18 (2020)
45. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: plug and play with succinct data structures. In: Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8504, pp. 326–337. Springer (2014)
46. Gog, S., Kärkkäinen, J., Kempa, D., Petri, M., Puglisi, S.J.: Fixed block compression boosting in FM-indexes: theory and practice. Algorithmica **81**(4), 1370–1391 (2019)
47. Gog, S., Moffat, A., Petri, M.: CSA++: fast pattern search for large alphabets. In: Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017, pp. 73–82. SIAM (2017)
48. Goto, K.: Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In: Prague Stringology Conference 2019, Prague, Czech Republic, August 26-28, 2019, pp. 111–125. Czech Technical University in Prague, Faculty of Infor-

mation Technology, Department of Theoretical Computer Science (2019)
49. Grabowski, S., Raniszewski, M.: Sampled suffix array with minimizers. Softw. Pract. Exp. **47**(11), 1755–1771 (2017)
50. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2), 378–407 (2005)
51. Gupta, M., Bendersky, M.: Information retrieval with verbose queries. In: Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, August 9-13, 2015, pp. 1121–1124. ACM (2015)
52. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press, United Kingdom (1997)
53. Henzinger, M.R.: Finding near-duplicate web pages: a large-scale evaluation of algorithms. In: SIGIR 2006: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Seattle, Washington, USA, August 6-11, 2006, pp. 284–291. ACM (2006)
54. Hon, T., Mars, K., Young, G., Tsai, Y.C., Karalius, J.W., Landolin, J.M., Maurer, N., Kudrna, D., Hardigan, M.A., Steiner, C.C., Knapp, S.J., Ware, D., Shapiro, B., Peluso, P., Rank, D.R.: Highly accurate long-read HiFi sequencing data for five complex genomes. Scientific Data **7**(1), 399 (2020)
55. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a time-and-space barrier in constructing full-text indices. SIAM J. Comput. **38**(6), 2162–2178 (2009)
56. Ilie, L., Navarro, G., Tinta, L.: The longest common extension problem revisited and applications to approximate string searching. J. Discrete Algorithms **8**(4), 418–428 (2010)
57. Jain, C., Rhie, A., Hansen, N., Koren, S., Phillippy, A.M.: Long-read mapping to repetitive reference sequences using winnowmap2. Nat. Methods **19**, 705–710 (2022)
58. Jiang, J., Versteeg, S., Han, J., Hossain, M.A., Schneider, J.G., Leckie, C., Farahmandpour, Z.: P-gram: positional n-gram for the clustering of machine-generated messages. IEEE Access **7**, 88504–88516 (2019)
59. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987)
60. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings, Lecture Notes in Computer Science, vol. 2089, pp. 181–192. Springer (2001)
61. Kempa, D., Kociumaka, T.: String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In: Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019, pp. 756–767. ACM (2019)
62. Kempa, D., Kociumaka, T.: Breaking the $O(n)$-barrier in the construction of compressed suffix arrays and suffix trees. In: Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023, pp. 5122–5202. SIAM (2023)
63. Kociumaka, T.: Minimal suffix and rotation of a substring in optimal time. In: 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel, *LIPIcs*, vol. 54, pp. 28:1–28:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
64. Kosolobov, D., Sivukhin, N.: Construction of sparse suffix trees and LCE indexes in optimal time and space. In: 35th Annual Symposium on Combinatorial Pattern Matching, CPM 2024, June 25-27, 2024, Fukuoka, Japan, LIPIcs, pp. 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024)
65. Kurtz, S.: Reducing the space requirement of suffix trees. Softw. Pract. Exp. **29**(13), 1149–1171 (1999)
66. Kärkkäinen, J., Kempa, D.: LCP array construction in external memory. ACM J. Exp. Algorithmics **21**(1), 1.7:1-1.7:22 (2016)
67. Kärkkäinen, J., Kempa, D.: LCP array construction using O(sort(n)) (or less) I/Os. In: S. Inenaga, K. Sadakane, T. Sakai (eds.) String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings, Lecture Notes in Computer Science, vol. 9954, pp. 204–217 (2016)
68. Kärkkäinen, J., Kempa, D.: Better external memory LCP array construction. ACM J. Exp. Algorithmics **24**(1), 1.3:1-1.3:27 (2019)
69. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Parallel external memory suffix sorting. In: Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings, Lecture Notes in Computer Science, pp. 329–342. Springer (2015)
70. Kärkkäinen, J., Kempa, D., Puglisi, S.J., Zhukova, B.: Engineering external memory induced suffix sorting. In: Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017, pp. 98–108. SIAM (2017)
71. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006)
72. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. **10**(3), R25 (2009)
73. Li, H., Durbin, R.: Fast and accurate short read alignment with burrows-wheeler transform. Bioinform. **25**(14), 1754–1760 (2009)
74. Li, R., Yu, C., Li, Y., Lam, T.W., Yiu, S.M., Kristiansen, K., Wang, J.: SOAP2: an improved ultrafast tool for short read alignment. Bioinform. **25**(15), 1966–1967 (2009)
75. Li, Z., Li, J., Huo, H.: Optimal in-place suffix sorting. Inf. Comput. **285**(Part), 104818 (2022)
76. Logsdon, G.A., Vollger, M.R., Eichler, E.E.: Long-read human genome sequencing and its applications. Nat. Rev. Genet. **21**(10), 597–614 (2020)
77. Loukides, G., Pissis, S.P.: Bidirectional string anchors: a new string sampling mechanism. In: 29th Annual European Symposium on Algorithms, ESA 2021, September6-8, 2021, Lisbon, Portugal (Virtual Conference), LIPIcs, pp. 64:1–64:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
78. Loukides, G., Pissis, S.P., Sweering, M.: Bidirectional string anchors for improved text indexing and top-K similarity search. IEEE Trans. Knowl. Data Eng. **35**(11), 11093–11111 (2023)
79. Maekawa, M.: A square root n algorithm for mutual exclusion in decentralized systems. ACM Trans. Comput. Syst. **3**(2), 145–159 (1985)
80. Manber, U., Myers, E.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
81. McCreight, E.M.: Priority search trees. SIAM J. Comput. **14**(2), 257–276 (1985)
82. Medelyan, O., Witten, I.H.: Thesaurus based automatic keyphrase indexing. In: ACM/IEEE Joint Conference on Digital Libraries, JCDL 2006, Chapel Hill, NC, USA, June 11-15, 2006, Proceedings, pp. 296–297. ACM (2006)
83. Morrison, D.R.: Patricia - practical algorithm to retrieve information coded in alphanumeric. J. ACM **15**(4), 514–534 (1968)
84. Munro, J.I., Navarro, G., Nekrich, Y.: Space-efficient construction of compressed indexes in deterministic linear time. In: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, pp. 408–424 (2017)

85. Mäkinen, V., Navarro, G.: Position-restricted substring searching. In: LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings, Lecture Notes in Computer Science, vol. 3887, pp. 703–714. Springer (2006)
86. Müller, I., Ratsch, C., Färber, F.: Adaptive string dictionary compression in in-memory column-store database systems. In: Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014, pp. 283–294. OpenProceedings.org (2014)
87. Navarro, G.: Compact Data Structures - A Practical Approach. Cambridge University Press, United Kingdom (2016)
88. Navarro, G., Nekrich, Y.: Time-optimal top-k document retrieval. SIAM J. Comput. **46**(1), 80–113 (2017)
89. Ohlebusch, E., Fischer, J., Gog, S.: CST++. In: String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings, Lecture Notes in Computer Science, vol. 6393, pp. 322–333. Springer (2010)
90. Roberts, M., Hayes, W., Hunt, B.R., Mount, S.M., Yorke, J.A.: Reducing storage requirements for biological sequence comparison. Bioinform. **20**(18), 3363–3369 (2004)
91. Rodriguez-Tomé, P., Stoehr, P., Cameron, G., Flores, T.P.: The european bioinformatics institute (EBI) databases. Nucleic Acids Res. **24**(1), 6–12 (1996)
92. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst. **41**(4), 589–607 (2007)
93. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: Local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, pp. 76–85. ACM (2003)
94. Umemoto, K., Song, R., Nie, J.Y., Xie, X., Tanaka, K., Rui, Y.: Search by screenshots for universal article clipping in mobile apps. ACM Trans. Inf. Syst. **35**(4), 34:1-34:29 (2017)
95. Vitter, J.S.: Algorithms and data structures for external memory. Found. Trends Theor. Comput. Sci. **2**(4), 305–474 (2006)
96. Vogelsgesang, A., Haubenschild, M., Finis, J., Kemper, A., Leis, V., Mühlbauer, T., Neumann, T., Then, M.: Get real: how benchmarks fail to represent the real world. In: Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018, pp. 1:1–1:6. ACM (2018)
97. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973, pp. 1–11. IEEE Computer Society (1973)
98. Wenger, A.M., et al.: Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. Nat. Biotechnol. **37**, 1155–1162 (2019)
99. Zheng, H., Kingsford, C., Marçais, G.: Improved design and analysis of practical minimizers. Bioinform **36**(Supplement–1), i119–i127 (2020)
100. Zheng, H., Marçais, G., Kingsford, C.: Creating and using minimizer sketches in computational genomics. J. Comput. Biol. **30**(12), 1251–1276 (2023)