Text Indexing for Long Patterns: Anchors are All you Need

Lorraine A. K. Ayad
Brunel University London
London, UK
lorraine.ayad@brunel.ac.uk

Grigorios Loukides
King's College London
London, UK
grigorios.loukides@kcl.ac.uk

Solon P. Pissis
CWI and Vrije Universiteit
Amsterdam, the Netherlands
solon.pissis@cwi.nl

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 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 the construction of the index proposed by Loukides and Pissis, which is based on bidirectional string anchors (bd-anchors), a new type of lc-anchors, by: **(i)** designing an average-case linear-time algorithm to compute bd-anchors; and **(ii)** developing a semi-external-memory implementation to construct the index in small space using near-optimal work. We then present an extensive experimental evaluation, based on the four measures, using real benchmark datasets. The results show that, for long patterns, the index constructed using our improved algorithms compares favorably to all classic indexes: (compressed) suffix tree; (compressed) suffix array; and the FM-index.

PVLDB Reference Format:
Lorraine A. K. Ayad, Grigorios Loukides, and Solon P. Pissis. Text Indexing for Long Patterns: Anchors are All you Need. PVLDB, 16(9): 2117 - 2131, 2023.
doi:10.14778/3598581.3598586

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lorrainea/BDA-index.

1 INTRODUCTION

In many real-world database systems, including bioinformatics systems [83], Enterprise Resource Planning (ERP) systems [75], or Business Intelligence (BI) systems [88], 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 [14] but also to *simultaneously* enable fast pattern matching queries. This is the classic text indexing problem [23, 46, 78]. 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 or not in S or *report* the set of all positions in S where an occurrence of P starts.

1.1 Motivation and Related Work

A considerable amount of algorithmic research has been devoted to text indexes over the past decades [5, 6, 22, 26, 28, 35, 44, 48, 57, 59, 68, 72, 76, 89]. This is mainly due to the fact that myriad string processing tasks (see [2, 46] 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* [89] and the latter is known as the *suffix array* [72]. 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 $\tilde{O}(|P|)$ time[1]. The time for reporting is $O(1)$ per occurrence for both structures. Thus, if a pattern P occurs occ times in S, the total **query time** for reporting is $\tilde{O}(|P| + \text{occ})$.

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 [26], who showed that suffix trees (and thus suffix arrays, indirectly) can be constructed in $O(n)$ time when Σ is an integer alphabet of size $\sigma = n^{O(1)}$. 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 $O(\log_\sigma n)$ 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 [62] is about 40GB, whereas the genome occupies less than 1GB. To address the above issue, Grossi and Vitter [44] and Ferragina and Manzini [28], and later Sadakane [84], introduced, respectively, the *compressed suffix array*

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

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 9 ISSN 2150-8097.
doi:10.14778/3598581.3598586

(CSA), the *FM-index*, and the *compressed suffix tree* (CST). These data structures occupy $O(n \log \sigma)$ bits, instead of $O(n \log n)$ bits, at the expense of a factor of $O(\log^{\epsilon} n)$ penalty in the query time, where $\epsilon > 0$ is an arbitrary predefined constant. These indexes have the dominant role in some of the most widely used bioinformatics tools [63–65]; while mature and highly engineered implementations of these indexes are available through the sdsl-lite library of Gog et al. [38].

Nowadays, as the data volume grows rapidly, **construction space** is as well becoming crucial for several string processing tasks [6, 60]. Suffix arrays can be constructed in optimal time using $O(1)$ words of extra space with the algorithm of Franceschini and Muthukrishnan [32] for general alphabets or with the algorithms of Goto [41] or Li et al. [66] for integer alphabets. *Extra* refers to the required space except for the space of S and the output. Grossi and Vitter's [44] original algorithm for constructing the CSA takes $O(n \log \sigma)$ time using $O(n \log n)$ bits of construction space. Hon et al. [48] showed an $O(n \log \log \sigma)$-time construction reducing the construction space to $O(n \log \sigma)$ bits. More recently, Belazzougui [5] and, independently, Munro et al. [76], improved the time complexity of the CSA/CST construction to $O(n)$ using a construction space of $O(n \log \sigma)$ bits. Very recently, Kempa and Kociumaka [60] have presented a new data structure that occupies $O(n \log \sigma)$ bits, can be constructed in $O(n \log \sigma / \sqrt{\log n})$ time using $O(n \log \sigma)$ 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* [72] 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* [44], the *CST* [84], or the *FM-index* [28] occupy $O(n \log \sigma)$ bits of space; they can be constructed very fast using $O(n \log \sigma)$ 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. For instance, in bioinformatics [50, 67, 90], the length of sequencing reads (patterns) ranges from a few hundreds to 30 thousand [67]. Even when at most k errors must be accommodated for matching, at least one out of $k+1$ fragments must be matched exactly. In natural language processing, the queried patterns can also be long [86]. Examples of such patterns are queries in question answering systems [45], *description queries* in TREC datasets [3, 10], and representative phrases in documents [73]. Similarly, a query pattern can be long when it encodes an entire document (e.g., a webpage in the context of deduplication [47]), or machine-generated messages [51].

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 any pattern P, $|P| \geq \ell$, will not 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 [68] 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 $O(n/\ell)$ in expectation [68, 69] and that can be constructed in $O(n)$ worst-case time [68]. 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 $\tilde{O}(n)$ time and can report all occ occurrences of any pattern P of length $|P| \geq \ell$ in S in $\tilde{O}(|P| + \text{occ})$ time (see Section 2 for a formal theorem). The authors of [68] have also implemented their index and presented some very promising experimental results (see also [69]).

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 $O(n)$-time worst-case algorithm seems too complicated to implement and unlikely to be efficient in practice [68]; **(ii)** their index construction implementation uses $\Theta(n)$ construction space *in any case*, which is much larger than the expected size $|\mathcal{A}_\ell(S)| = O(n/\ell)$ of the index.

Here, we improve on the construction of the index proposed by Loukides and Pissis [68] by addressing these aspects as follows:

- We design a novel average-case $O(n)$-time algorithm to compute $\mathcal{A}_\ell(S)$. To achieve this, we use minimizers [82, 85], another well-known type of lc-anchors, as anchors, to compute $\mathcal{A}_\ell(S)$ after carefully setting the sampling parameters. We employ longest common extension (LCE) queries [59] on S to compare anchored rotations (i.e., rotations of S starting at minimizers) of fragments of S efficiently. The fact that minimizers are $O(n/\ell)$ in expectation lets us realize the average-case $O(n)$-time computation of $\mathcal{A}_\ell(S)$.
- We propose a semi-external-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 $O(\ell)$ space using the aforementioned algorithm. Then 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 $O(\ell + |\mathcal{A}_\ell(S)|)$ space.

We present an extensive experimental evaluation using real benchmark datasets. First, we show that our new algorithm for computing $\mathcal{A}_\ell(S)$ 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 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 our improved 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 30% faster query time, for all $\ell \in [16, 1024]$, 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 8 times less space, for $\ell = 1024$, 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. As another example, our index on the full human genome occupies 16MB for $\ell = 2^{14}$ and answers queries more than 32 times faster than the FM-index, which occupies more than 1GB.

1.3 Other Related Work

The connection of lc-anchors to text indexing and related applications is not new. The perhaps most popular lc-anchors in practical applications are *minimizers*, which have been introduced independently by Schleimer et al. [85] and by Roberts et al. [82] (see Section 2 for a definition). Although minimizers have been mainly used for sequence comparison [50], Grabowski and Raniszewski [42] showed how minimizers can be used to sample the suffix array – see also [21], which uses an alphabet sampling approach. Another well-known notion of lc-anchors is *difference covers*, which have been introduced by Burkhardt and Kärkkäinen [15] for suffix array construction in small space (see also [70]). Difference covers play also a central role in the elegant linear-time suffix array construction algorithm of Kärkkäinen et al. [57]; and they have been used in other string processing applications [9, 17]. Another very powerful type of lc-anchors is *string synchronizing sets*, which have been recently proposed by Kempa and Kociumaka [59] for constructing, among others, an optimal data structure for LCE queries (see also [25]). String synchronizing sets have applications in designing sublinear-time algorithms for classic string problems [18, 19], whose textbook linear-time solutions rely on suffix trees.

1.4 Paper Organization

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 [68]. In Section 4, we improve the construction of this index by presenting: **(i)** our fast algorithm for bd-anchors computation (see Section 4.1); and **(ii)** the index construction in small space using near-optimal work (see Section 4.2). In Section 5, we provide the full details of our implementations, and in Section 6, we present our extensive experimental evaluation. We conclude this paper in Section 7.

2 PRELIMINARIES

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

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

Classic Text Indexes. For a string S of length n over an ordered alphabet of size σ, the *suffix array* $\mathsf{SA}[1\,.\,.\,n]$ stores the permutation of $\{1, \dots, n\}$ 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 $P[1\,.\,.\,m]$, known as the *pattern*, the suffix array of S allows us to report all occ occurrences of P in S using only $O(m \log n + \text{occ})$ operations [72]. We perform binary search in SA resulting in a range $[s, e)$ of suffixes of S having P as a prefix. Then, $\mathsf{SA}[s\,.\,.\,e-1]$ contains the starting positions of all occurrences of P in S. The SA is often augmented with the LCP array [72] 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 $O(m + \log n + \text{occ})$ time by avoiding to compare P with suffixes of S from scratch during binary search [72] (see [22, 30, 79] for subsequent improvements). The suffix array occupies $\Theta(n)$ space and it can be constructed in $O(n)$ time for an integer alphabet of size $\sigma = n^{O(1)}$ [26]. Given SA of S, we can compute the LCP array of S in $O(n)$ time [58].

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 $O(|\mathcal{F}|)$ space [74]. 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 [89]. Assuming S ends with a unique terminating symbol, every leaf in $\mathsf{ST}(S)$ represents a suffix $S[i\,.\,.\,n]$ 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 $S[i\,.\,.\,n]$. 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 $P[1\,.\,.\,m]$, $\mathsf{ST}(S)$ allows us to report all occ occurrences of P in S using only $O(m \log \sigma + \text{occ})$ 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 $O(n)$ time for an integer alphabet of size $\sigma = n^{O(1)}$ [26]. To improve the query time to the optimal $O(m + \text{occ})$ we can use randomization to construct a perfect hash table [33] for accessing edges by the first letter of their label in $O(1)$ 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 $\sigma = n^{O(1)}$, we can construct a data structure in $O(n/\log_\sigma n)$ time that answers $\mathsf{LCP}_S(i, j)$ queries in $O(1)$ time [59].

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 [82]. 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 $S[i \mathinner{.\,.} i + w + k - 2]$, for all $i \in [1, n - w - k + 2]$.

Example 2.1 (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 2.2 ([91]). *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 $O(n/w)$ if and only if $k \geq \log_\sigma w + O(1)$.*

Lemma 2.3 ([68]). *For any string S of length n over an integer alphabet of size $n^{O(1)}$ and integers $w, k > 0$, the set $\mathcal{M}_{w,k}$ can be computed in $O(n)$ time.*

Loukides and Pissis defined bidirectional anchors (bd-anchors) as an alternative notion of lc-anchors [68].

Definition 2.4 (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 $S[i \mathinner{.\,.} i + \ell - 1]$, for all $i \in [1, n - \ell + 1]$.

Example 2.5 (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. [69] according to the following definition.

Definition 2.6 (Reduced bidirectional 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 $S[i \mathinner{.\,.} i + \ell - 1]$, for all $i \in [1, n - \ell + 1]$.

Informally, when using the reduced bd-anchor mechanism, we neglect the r rightmost rotations from the sampling process.

Example 2.7 (Reduced bd-anchors). Let $S = \texttt{aacaaacgcta}$, $\ell = 5$, and $r = 1$. Recall from Example 2.5 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 2.8 ([68, 69]). *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 $O(n/\ell)$.*

Lemma 2.9 ([68, 69]). *For any string S of length n over an integer alphabet of size $n^{O(1)}$ and integers $\ell > 0, r \geq 0$, the set $\mathcal{A}_{\ell,r}(S)$ can be computed in $O(n)$ time.*

3 THE INDEX

Loukides and Pissis [68] proposed the following text index, which is based on (reduced [69]) bd-anchors. Fix string S of length n over an integer alphabet of size $n^{O(1)}$ as well as ℓ (and r). Given $\mathcal{A}_{\ell,r}(S)$ we construct two compacted tries: one for strings $S[i \mathinner{.\,.} n]$, for all $i \in \mathcal{A}_{\ell,r}(S)$, which we denote by $\mathcal{T}^R_{\ell,r}(S)$; and one for strings $\overleftarrow{S[1 \mathinner{.\,.} i]}$, for all $i \in \mathcal{A}_{\ell,r}(S)$, which we denote by $\mathcal{T}^L_{\ell,r}(S)$. We also

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

(b) Querying $P =$ acaaa.

Figure 1: 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 $P[1 \mathinner{.\,.} \ell]$ for $\ell = 5$ and $r = 1$, which is $j = 3$: the minimal lexicographic rotation is aaaac. We then split P in $\overleftarrow{P[1 \mathinner{.\,.} 3]} =$ aca and $P[3 \mathinner{.\,.} |P|] =$ aaa, and search $\overleftarrow{P[1 \mathinner{.\,.} 3]} =$ aca in $\mathcal{T}^L_{\ell,r}$ and $P[3 \mathinner{.\,.} |P|] =$ 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, $P = S[2 \mathinner{.\,.} 5] =$ acaaa: the fragment $S[2 \mathinner{.\,.} 5]$ is anchored at position 4.

construct a 2D range reporting data structure [16] over $|\mathcal{A}_{\ell,r}(S)|$ points (x, y), where x is the lexicographic rank of $S[i \mathinner{.\,.} n]$ and y is the lexicographic rank of $\overleftarrow{S[1 \mathinner{.\,.} i]}$, for all $i \in \mathcal{A}_{\ell,r}(S)$. Upon a query P of length $|P| \geq \ell$, we find the (reduced) bd-anchor of $P[1 \mathinner{.\,.} \ell]$, say j, which gives $P[j \mathinner{.\,.} |P|]$ and $\overleftarrow{P[1 \mathinner{.\,.} j]}$. We search the former in $\mathcal{T}^R_{\ell,r}$ and the latter in $\mathcal{T}^L_{\ell,r}$. This search induces a rectangle (inspect Figure 1, for an example), which we use to query the 2D range reporting data structure. The reported points are all occurrences of P in S. We denote the resulting index by $\mathcal{I}_{\ell,r}(S)$. Loukides and Pissis showed the following result [68, 69].

Theorem 3.1 ([68, 69]). *Given any string S of length n over an integer alphabet of size $n^{O(1)}$ and integers $\ell > 0, r \geq 0$, the index $\mathcal{I}_{\ell,r}(S)$ occupies $O(|\mathcal{A}_{\ell,r}(S)|)$ extra space and reports all occ occurrences of any pattern P of length $|P| \geq \ell$ in S in $\tilde{O}(|P| + occ)$ time. Moreover, the index $\mathcal{I}_{\ell,r}(S)$ can be constructed in $\tilde{O}(n)$ 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 [4, 7], where $H_0(S)$ is the *zeroth-order entropy* of S. Note that the same index can be constructed for any lc-anchors; e.g., for minimizers.

4 IMPROVING THE INDEX

In this section, we improve the construction of the index $\mathcal{I}_{\ell,r}(S)$ (Theorem 3.1) focusing on two practical aspects: **(i)** the fast computation of reduced bd-anchors (see Section 4.1); and **(ii)** the index construction in small space using near-optimal work (see Section 4.2). With *small*, we mean space that is close to the index size.

4.1 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 [68, 69] showed a worst-case linear-time algorithm to compute $\mathcal{A}_{\ell,r}(S)$ (Lemma 2.9). 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 [61]. The latter data structure relies heavily on the construction of *fusion trees* [34], and hence it is mostly of theoretical interest: it is widely accepted that naïve solutions can be more practical than such sophisticated data structures – see [24] 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 [68]. 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 $O(n\ell)$ time as the simple algorithm from [68] does.

Main idea. We use (w, k)-minimizers as anchors to compute reduced bd-anchors of order ℓ after carefully setting parameters w and k. We employ $\text{LCP}_S(i, j)$ 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 $O(n/\ell)$ in expectation lets us realize $O(n)$ 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 minimal rotation of F with minimal $j \in [1, \ell - r]$. By definition, $F[j \mathinner{.\,.} |F|]$ is of length at least $r + 1$ and there cannot be another j' such that $F[j' \mathinner{.\,.} j' + r]$ is lexicographically smaller than $F[j \mathinner{.\,.} j + r]$. Since $\ell = w + k - 1 \implies w = \ell - r$, j is an $(\ell - r, r + 1)$-minimizer of F. ◻

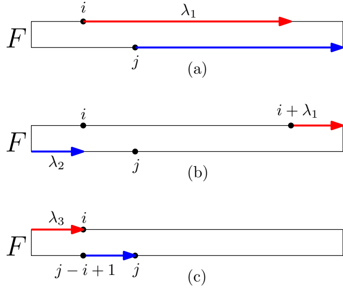


Figure 2: Illustration of Lemma 4.2.

Example 4.1 (Cont'd from Example 2.7). 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, $\mathcal{A}_{5,1}(S) = \{4, 5, 6, 7\} \subseteq \mathcal{M}_{4,2}(S) = \{1, 4, 5, 6, 7\}$.

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 2.3) to compute reduced bd-anchors by setting $w = \ell - r$ and $k = r+1$. The following lemma is crucial for the efficiency of our algorithm. 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 4.2. *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.*

Proof. Assume without loss of generality that $i < j$ (inspect Figure 2). We first ask for the length λ_1 of the LCP of $F[i \,..\, |F|]$ and $F[j \,..\, |F|]$. If $\lambda_1 < |F| - j + 1$, then we simply compare letters $F[i+\lambda_1]$ and $F[j+\lambda_1]$ 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 letters $F[i + \lambda_1 + \lambda_2]$ and $F[1+\lambda_2]$ 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 letters $F[1 + \lambda_3]$ and $F[j - i + 1 + \lambda_3]$ to find the answer. Otherwise, rotation i of F is equal to rotation j of F (see bottom part (c)). ◻

We next use Lemma 4.2 as a building block to obtain Lemma 4.3, the main lemma used by our algorithm.

Lemma 4.3. *Let D be a string of length $|D| \geq \ell$, for some integer $\ell > 0$. 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 $j \in A : j \in [i, i+\ell-1]$. Given a data structure for answering LCP_D queries in $O(1)$ time, we can find the minimal lexicographic rotation j in every length-ℓ fragment of D, such that $j \in A$ and j is minimal in $O(d|D|)$ total time.*

Proof. First we sort the elements of A in increasing order using radix sort in $O(|D|)$ 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. For any two elements, we perform an application of Lemma 4.2, which takes $O(1)$ time, and maintain the leftmost lexicographically minimal rotation. Inspect Figure 3.

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 $O(d|D|)$. ◻

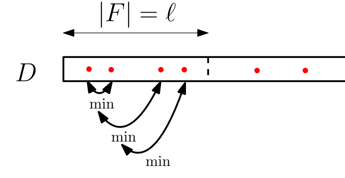


Figure 3: Illustration of three applications of Lemma 4.2 on F. We mark the elements of A in red circles.

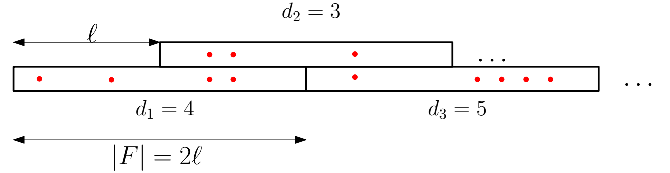


Figure 4: Illustration of the decomposition of S. We mark the (w, k)-minimizers in red circles.

Theorem 4.4. *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 $O(n/\ell)$. Moreover, if Σ is an integer alphabet of size $\sigma = n^{O(1)}$, $\mathcal{A}_{\ell,r}(S)$ can be computed in $O(n)$ time on average.*

Proof. The first part is merely Lemma 2.8 from [69].

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 $O(n)$ time by Lemma 2.3 [68]. We also construct an LCP_S data structure in $O(n)$ time [59]. 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 4.3 with $D = F_i, d = d_i$ and $A = \mathcal{M}_{w,k}(F_i)$. This takes $d_i|F_i| = O(d_i\ell)$ time. Note that $q = \Theta(n/\ell)$. Then the total work of the algorithm, by Lemma 4.3, is bounded *on average* by:

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

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 2.2, 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 4.4 takes $\Theta(n\ell)$ time.

4.2 Index Construction in Small Space

A straightforward implementation of the index $\mathcal{I}_{\ell,r}(S)$ 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 2.3 [68]; and **(ii)** the construction of $\mathcal{T}^R_{\ell,r}(S)$ and $\mathcal{T}^L_{\ell,r}(S)$ [68].

However, the size of $\mathcal{I}_{\ell,r}(S)$ can be asymptotically smaller than $\Theta(n)$. In fact, by Theorem 3.1 the size of $\mathcal{I}_{\ell,r}(S)$ is $O(|\mathcal{A}_{\ell,r}(S)|)$, which is expected to be $O(n/\ell)$. Note that $|\mathcal{A}_{\ell,r}(S)| = \Omega(n/\ell)$ in any case [68], 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 $O(\ell + |\mathcal{A}_{\ell,r}(S)|)$ space (and external memory). (Note that in practical applications ℓ does not exceed n/ℓ and so ℓ is negligible.) In the standard *external memory* (EM) model [87], with internal memory (RAM) size M and disk block size B, the standard I/O complexities are: $\text{scan}(n) = n/B$, which is the complexity of scanning n elements sequentially; and $\text{sort}(n) = (n/B)\log_{M/B}(n/B)$, which is the complexity of sorting n elements [87]. *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 $O(\ell)$ extra space; in Step 2, we compute the SA and LCP array of S (and $\overleftarrow{S}$) 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. We next describe in detail how every step is implemented using near-optimal work.

Step 1. We make use of $O(\ell)$ extra space. Specifically, we use Theorem 4.4 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 with respect to $|F|$), we use $\tilde{O}(\ell)$ time instead of $O(\ell)$ by first sorting the letters of F, and then assigning each letter to a rank in $\{1,\dots,2\ell\}$ accordingly. This clearly does not affect the lexicographic rank of rotations. Thus the total time to construct $\mathcal{A}_{\ell,r}(S)$ is $\tilde{O}(n)$ on average: there are $O(n/\ell)$ fragments and each one is processed in $\tilde{O}(\ell)$ time. We finally implement $\mathcal{A}_{\ell,r}(S)$ as a perfect hash table, denoted by $\mathcal{H}_{\ell,r}(S)$, which we keep in RAM. This is done in $O(|\mathcal{A}_{\ell,r}(S)|)$ time supporting $O(1)$-time look-ups [33].

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 [12, 57]; these algorithms are optimal with respect to internal work $O(n\log_{M/B}(n/B))$ and I/O complexity $O((n/B)\log_{M/B}(n/B))$—see also [52–56]. We also compute the SA and LCP array of $\overleftarrow{S}$, the reverse of S, analogously.

Step 3. In this step we will 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 $\text{RSA}[i]$ is the starting position of the ith lexicographically smallest suffix of S with $\text{RSA}[i] \in \mathcal{A}_{\ell,r}(S)$. $\text{RLCP}[i]$ array (Right LCP array) stores the length of the LCP of $\text{RSA}[i-1]$ and $\text{RSA}[i]$. LSA and LLCP array are defined analogously for $\overleftarrow{S}$. Let us show how we construct RSA and RLCP array; 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 $\text{SA}[i]$ and $\text{LCP}[i]$. This is possible using $O(1)$ words of extra memory. If $\text{SA}[i]$ is in $\mathcal{H}_{\ell,r}(S)$, which we check in $O(1)$ time, we set $\text{RSA}[k] = \text{SA}[i]$. It is also well known that for any $i_1 < i_2$ the length of the LCP between $S[\text{SA}[i_1]\,..\,n]$ and $S[\text{SA}[i_2]\,..\,n]$ is the minimum value lying in $\text{LCP}[i_1+1],\dots,\text{LCP}[i_2]$. Since we scan also the LCP array simultaneously, we maintain the value we need to store in $\text{RLCP}[k]$. Finally we increment k and i by one. Scanning RSA and RLCP array takes $O(n/B)$ I/Os. Using RSA and RLCP array we can construct $\mathcal{T}^R_{\ell,r}(S)$ in $O(|\mathcal{A}_{\ell,r}(S)|)$ time using a folklore algorithm (cf. [58]). We construct $\mathcal{T}^L_{\ell,r}(S)$ analogously from LSA and LLCP array.

Step 4. By using RSA and LSA, we construct the 2D range reporting data structure in $\tilde{O}(|\mathcal{A}_{\ell,r}(S)|)$ time using $O(|\mathcal{A}_{\ell,r}(S)|)$ space [8, 16, 36, 71].

This completes the construction. Indeed, we have shown how to construct $\mathcal{I}_{\ell,r}(S)$ in near-optimal work using only $O(\ell + |\mathcal{A}_{\ell,r}(S)|)$ space and external memory. Let us remark that Steps 2-3 can be implemented in $\tilde{O}(n)$ time using $O(|\mathcal{A}_{\ell,r}(S)|)$ space assuming read-only random access to S [13, 37, 49, 81]. The reason we focus on the (semi-)EM approach is because it is much more developed implementation-wise [12, 52, 54] than sparse suffix sorting.

5 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 [35], 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 [42], as their number is generally greater than (reduced) bd-anchors; see the results of [68, 69]. As 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. Although many other engineered implementations for classic text indexes exist, e.g. [39] for FM-index or [11] for suffix array, we have based our implementations on sdsl-lite [38] as much as possible for fairness:

- **SA:** The suffix array was constructed using the `divsufsort` class, written by Yuta Mori, as included in sdsl-lite. To report all occurrences of a pattern, we implemented the algorithm of Manber and Myers [72] that uses as well the LCP array [58] augmented with a succinct RMQ data structure [31] (`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 [80] 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 open-source implementation of the r-index [35] was used. The binary `ri-build` was used to build the index using the `divsufsort` class of the sdsl-library and `ri-locate` used to report all occurrences of a pattern.

BDA-compute: We implemented the average-case linear-time algorithm for computing the set of reduced bd-anchors as described in Section 4.1. To improve construction space, we implemented the algorithm in fragments as described in Step 1 of the space-efficient algorithm presented in Section 4.2. The implementation takes the fragment length b as input. We call these fragments *blocks* henceforth.

BDA-index I: The construction was implemented using Steps 1 to 4 as described in Section 4.2. For Step 2, we used the pSAscan algorithm of Kärkkäinen et al. [55] to construct the SA in EM and the EM-SparsePhi algorithm of Kärkkäinen and Kempa [54] 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. For Step 4, we implemented the 2D range reporting data structure of Mäkinen and Navarro [71]. To report all occurrences of a pattern, we implemented the algorithm of Loukides and Pissis [68] (see Section 3).

BDA-index II: The construction was implemented using Steps 1 to 3 as in BDA-index I. Unlike BDA-index I, however, no 2D range reporting data structure was used (Step 4). We instead used the bidirectional search algorithm presented by Loukides and Pissis in [68]. 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 [68]; note, however, the query time of this index is not bounded.

6 EXPERIMENTS

6.1 Experimental Setup

We considered five datasets (texts) from the Pizza&Chili corpus [27] – see Table 1 for their characteristics. We generated patterns of length 16, 32, 64, 128, 256, 512, and 1024, for all datasets, with 500,000 patterns created per length. The patterns were generated by selecting occurrences uniformly at random from the datasets. We also considered the *Homo sapiens* genome (version GRCh38.p14), which we denote by HUMAN. Its length n is 3,136,895,129 and the alphabet size σ is 30. We generated patterns of length 64, 256, 1,024, 4,096, and 16,384 for HUMAN with 500,000 patterns created per length. Similar to the datasets of Table 1, these patterns were generated by selecting occurrences uniformly at random from HUMAN.

The experiments ran on an Intel Xeon Gold 648 at 2.5GHz with 192GB RAM. All programs were compiled with g++ version `9.4.0` at the `-O3` optimization level. Our code and datasets are freely available at https://github.com/lorrainea/BDA-index.

Table 1: Datasets characteristics.

Dataset	Length n	Alphabet size σ
DNA	200,000,000	15
PROTEINS	200,000,000	24
XML	200,000,000	94
SOURCES	200,000,000	224
ENGLISH	200,000,000	221

Parameters. For BDA-compute, BDA-index I and BDA-index II, we set ℓ to the pattern length, b to 25K (unless stated otherwise), and r to $\lceil 4 \log \ell / \log \sigma \rceil$; we set M (RAM) to $\Theta(|\mathcal{A}_{\ell,r}|)$ as this is anyway 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 6.2 to 6.7 we present the results for the datasets of Table 1. In Section 6.8 we present the results for the HUMAN dataset but only for the FM-index, the r-index, and the BDA-index I and II (all the other indexes were not competitive).

6.2 Computing Bd-anchors

Figure 5 shows the time required to compute the reduced bd-anchors for the five datasets of Table 1. The $\Theta(n\ell)$ line represents the simple $\Theta(n\ell)$-time algorithm of [68]. The results show that for all five datasets, all b values, and all ℓ values, BDA-compute outperforms the $\Theta(n\ell)$-time implementation. In particular, BDA-compute becomes more than *two orders of magnitude faster* as ℓ increases. As suggested by Theorem 4.4, the runtime of BDA-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 BDA-compute in Figure 5 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 4.3), and this is why the runtime increases with ℓ. Nevertheless, even in this bad case, BDA-compute is up to 8× faster than the implementation of the simple $\Theta(n\ell)$-time algorithm [68].

Figure 6 shows the space required to construct the reduced bd-anchors for the five datasets of Table 1. For all datasets, all b values, and all ℓ values, BDA-compute performs *similarly* to the $\Theta(n\ell)$-time implementation, using just slightly more space as b increases. This is expected because the latter implementation uses only $\Theta(\ell)$

space [68] and $b \gg \ell$ in our experiments. These space-time results establish the practical usefulness of our Theorem 4.4.

6.3 Index Size

Figure 7 shows the index size for the five datasets of Table 1 when using the implementations of Section 5. As expected, the index size of BDA-index I and II decreases with increasing ℓ. Notably, for all datasets and $\ell \geq 64$, BDA-index I and II are smaller than all other indexes except for the FM-index. When $\ell \geq 512$, BDA-index I and II *outperform all indexes* for all datasets.

6.4 Query Time

Figure 8 shows the average query time for all datasets when using the implementations of Section 5 with $\ell = |P|$. For all datasets and $|P| \geq 64$, BDA-index I and II are up to *several orders of magnitude* faster than the compressed indexes, especially for large alphabets, which is consistent with the observations made in [29, 40]. Notably, for all datasets and ℓ values, BDA-index I and II are even faster than the SA. FM-index did not finish for $|P| = 16$ within 24 hours for XML. CST did not finish within 24 hours for $|P| \geq 64$, in the case of SOURCES and ENGLISH.

6.5 Index Construction Space

Figure 9 shows the index construction space required for the five datasets of Table 1 when using the implementations of Section 5. As expected, the construction space required by BDA-index I and II decreases with increasing ℓ. For all datasets and ℓ values, BDA-index II outperforms BDA-index I (as expected). Notably, for all datasets when $\ell \geq 128$, BDA-index I and II outperform all other indexes.

6.6 Index Construction Time

Figure 10 shows the time required to construct the index for the five datasets of Table 1 when using the implementations of Section 5. We set $b = 25$K for BDA-index I and II. For all datasets and ℓ values, BDA-index I and II outperform CSA and are outperformed by the FM-index, SA, r-index, and CST. This is, however, expected due to the construction of SA and LCP array in EM. BDA-index I and II are around 8× slower than the fastest to construct FM-index.

6.7 Index Construction in Internal Memory

Figure 11 shows the index construction space required for the five datasets of Table 1 when using an internal memory implementation to compute the SA and LCP array for BDA-index I and II. We set $b = 25$K for both BDA-index I and II. The results generally show that BDA-index I and II are outperformed by the existing indexes. This is, however, expected; it in fact highlights the motivation for this work and, in particular, one of our main contributions: the index construction in small space. As mentioned in Section 4.2, if one would like to have a near-linear-time construction in small space using only internal memory, sparse suffix sorting could be directly employed to replace the current external-memory construction.

Figure 12 shows the time required to construct the index for the five datasets of Table 1 when using an internal memory implementation to compute the SA and LCP array for BDA-index I and II. We set $b = 25$K for BDA-index I and II. The results generally show that as ℓ increases BDA-index I and II become competitive to the existing indexes (expected). An exception to this is with SOURCES (see Section 6.2 for an explanation).

Figure 13 shows the time required to construct BDA-index I and II for the five datasets of Table 1 for varying size of internal memory. An internal memory size ranging from 128MB to 2048MB was used. We set $b = 25K$ and $\ell = 128$ for both implementations. The results show that in general for all datasets, the construction time decreases as the size of the internal memory used increases.

6.8 The Four Measures on Human Genome

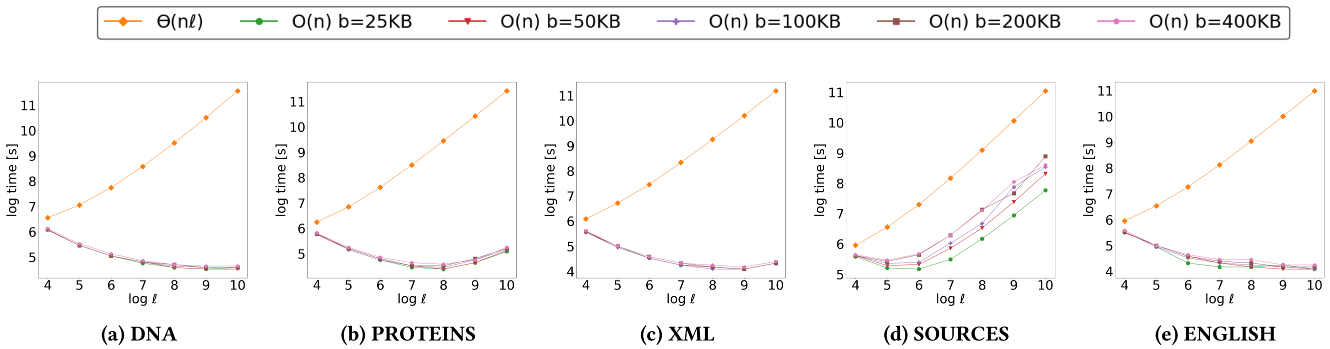
Figure 14a shows the index size for HUMAN when using BDA-index I, BDA-index II, FM-index and r-index. The index size occupied by BDA-index I and II decreases with increasing ℓ. For all ℓ, BDA-index I and II are smaller than r-index; and, for $\ell \geq 256$, BDA-index II is smaller than FM-index. For $\ell = 2^{14}$, BDA-index I and II take about 16MB, while FM-index and r-index take 1GB and 16GB, respectively. Figure 14b shows the average query time for HUMAN when using BDA-index I, BDA-index II, FM-index and r-index with $\ell = |P|$. As $|P|$ increases, BDA-index I and II become more than one order of magnitude faster than the FM-index and r-index. For all $|P|$, BDA-index II is faster than the FM-index and r-index and likewise for BDA-index I when $|P| \geq 256$. Figure 14c shows the index construction space for HUMAN when using BDA-index I, BDA-index II, FM-index and r-index. The construction space required by BDA-index I and II decreases with increasing ℓ. For all ℓ, BDA-index II requires less space than the FM-index and r-index and likewise for BDA-index I when $\ell \geq 256$. Figure 14d shows the time required to construct the index for HUMAN when using BDA-index I, BDA-index II, FM-index and r-index. We set $b = 130K$ for BDA-index I and II. For all ℓ, BDA-index I and II are around 8× and 4× slower than the FM-index and r-index, respectively.

The presented results highlight the scalability of our algorithms as well as the superiority of BDA-index I and II for long patterns.

7 DISCUSSION

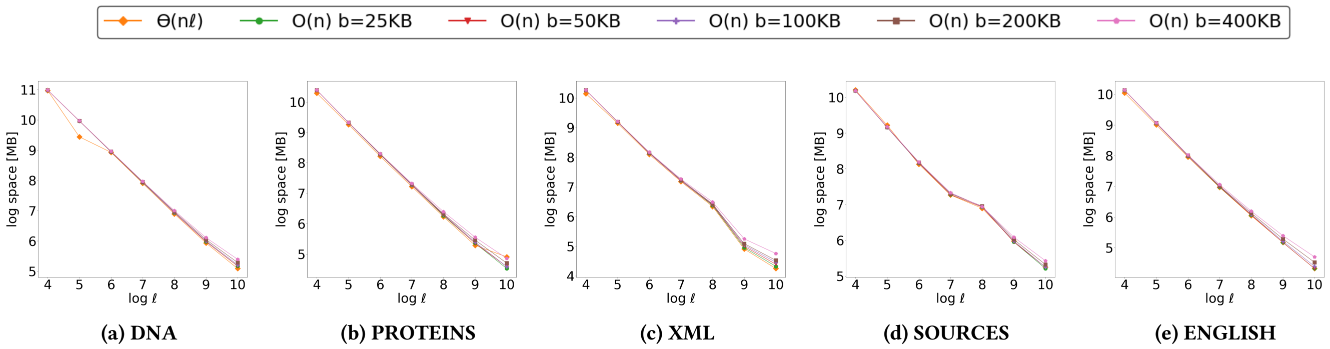
We have shown that our implementation of the bd-anchors index should be the practitioners' choice for long patterns. When $\ell \geq 512$, our index outperforms all indexes for all datasets in terms of index size. When $\ell \geq 64$, our index outperforms all indexes for all datasets in terms of query time. When $\ell \geq 128$, our index outperforms all indexes for all datasets in terms of construction space. The construction time of our index is between 4 and 8× slower than the state of the art. However, we believe that this does not make our index less practical; the construction time of the bd-anchors index can be improved in practice in at least two ways, which we describe below as future work (see the Future work paragraph).

An obvious criticism against our 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. First, we have shown, using real-world 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. Second, 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. Last, the state-of-the-art compressed indexes also have a crucial



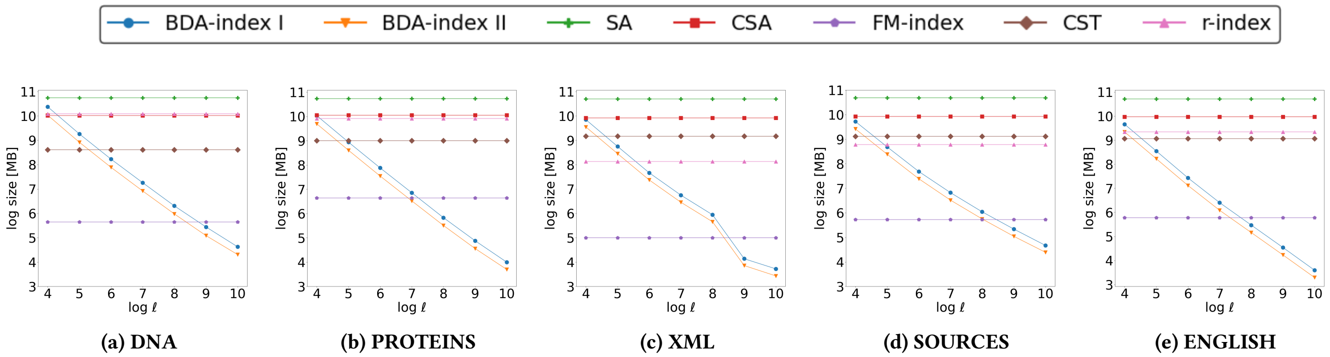
(a) DNA (b) PROTEINS (c) XML (d) SOURCES (e) ENGLISH

Figure 5: Elapsed time to construct the set of reduced bd-anchors (seconds in log-scale) for varying ℓ (log-scale).



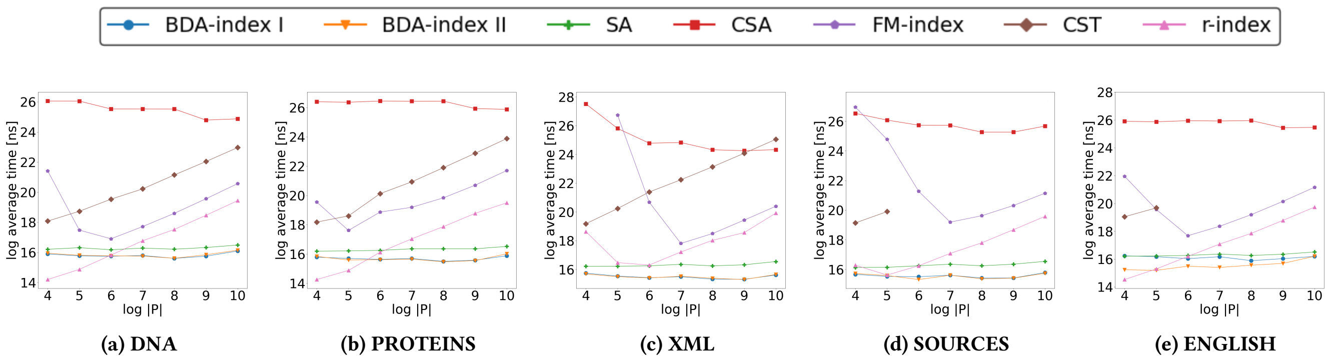
(a) DNA (b) PROTEINS (c) XML (d) SOURCES (e) ENGLISH

Figure 6: Space required to construct the set of reduced bd-anchors (MB in log-scale) for varying ℓ (log-scale).



(a) DNA (b) PROTEINS (c) XML (d) SOURCES (e) ENGLISH

Figure 7: Size of different indexes (MB in log-scale) for varying ℓ (log-scale).



(a) DNA (b) PROTEINS (c) XML (d) SOURCES (e) ENGLISH

Figure 8: Average time for pattern matching (nanoseconds in log-scale) for varying $|P|$ (log-scale).

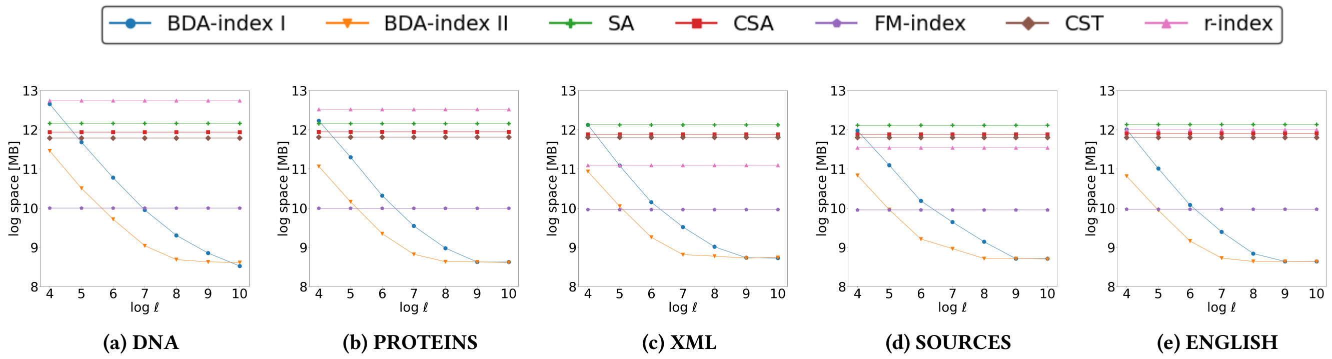


Figure 9: Space required to construct different indexes (MB in log-scale) for varying ℓ (log-scale) when using external memory.

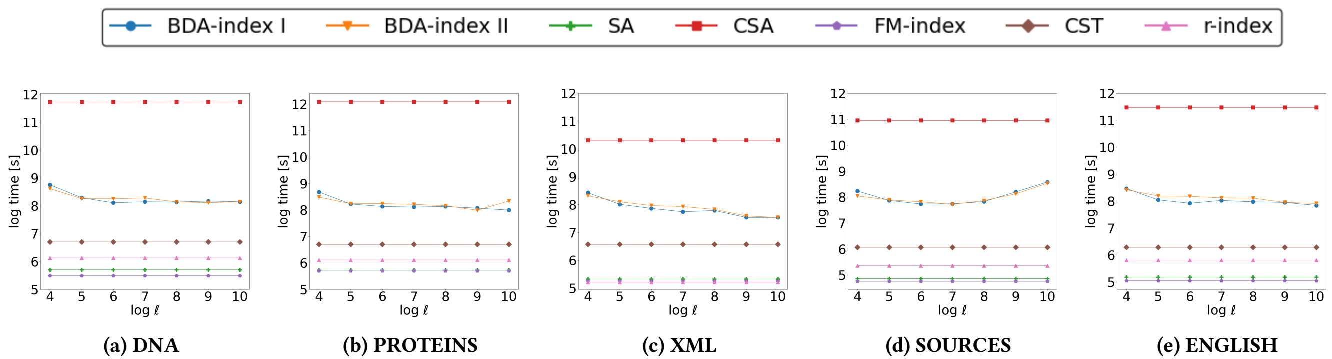


Figure 10: Elapsed time to construct different indexes (seconds in log-scale) for varying ℓ (log-scale) when using external memory.

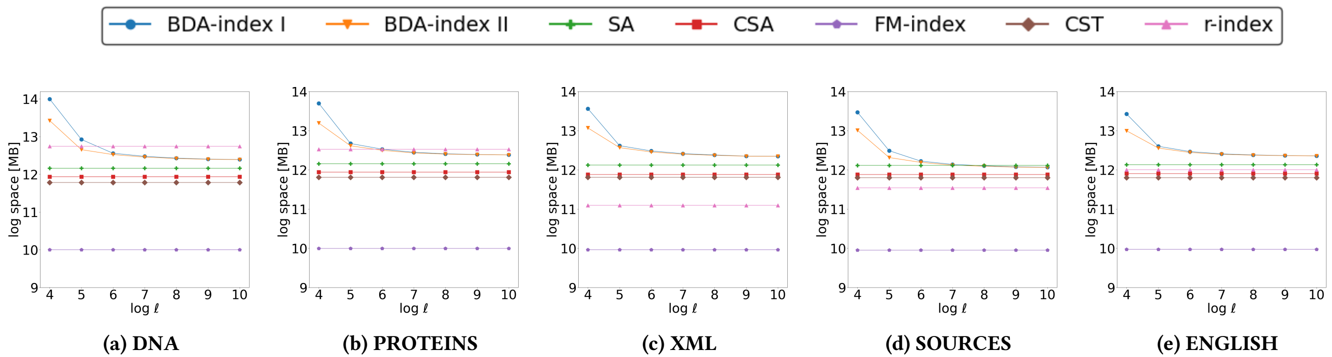


Figure 11: Space required to construct different indexes (MB in log-scale) for varying ℓ (log-scale) when using internal memory.

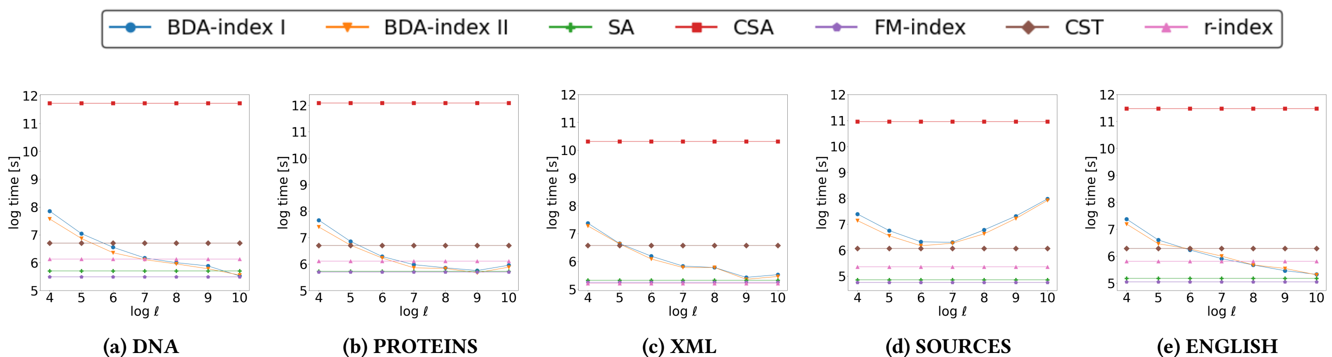
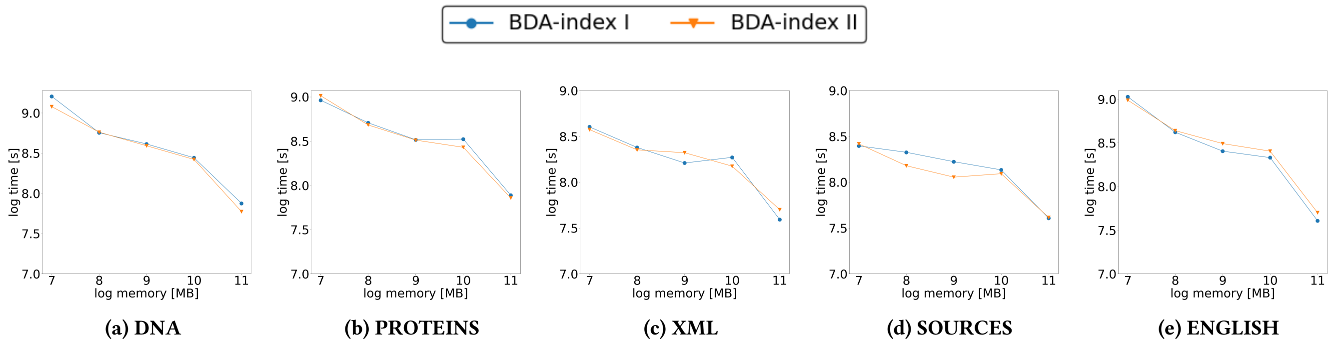
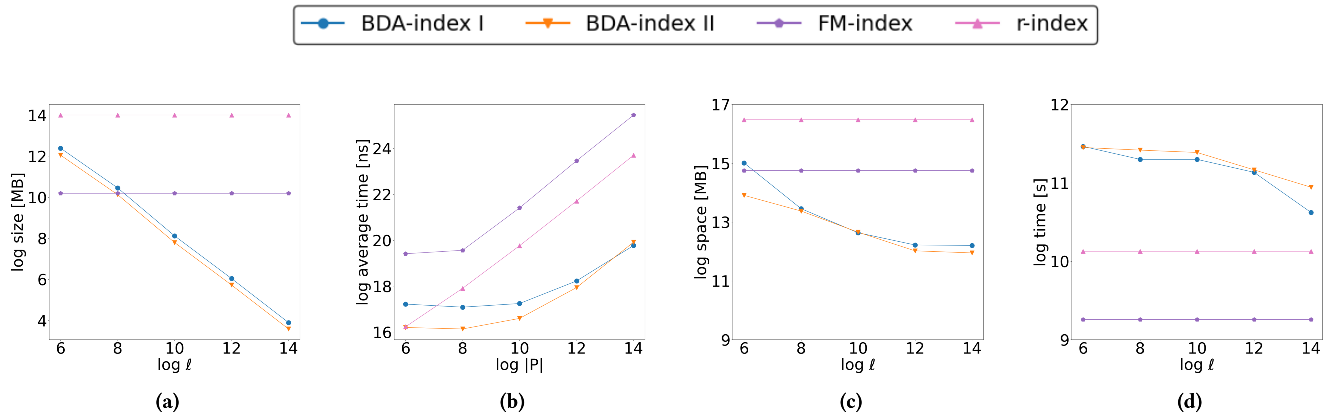


Figure 12: Elapsed time to construct different indexes (seconds in log-scale) for varying ℓ (log-scale) when using internal memory.



(a) DNA (b) PROTEINS (c) XML (d) SOURCES (e) ENGLISH

Figure 13: Elapsed time to construct BDA-index I and II (seconds in log-scale) using varying sizes of internal memory (MB in log-scale) for $\ell = 128$.



(a) (b) (c) (d)

Figure 14: (a) Size of different indexes (MB in log-scale) for HUMAN and varying ℓ (log-scale). (b) Average time for pattern matching (nanoseconds in log-scale) on HUMAN and for varying $|P|$ (log-scale). (c) Space required to construct different indexes (MB in log-scale) for HUMAN and varying ℓ (log-scale). (d) Elapsed time to construct different indexes (seconds in log-scale) for HUMAN and varying ℓ (log-scale).

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: they occupy $n \log \sigma = \Theta(n \log n)$ bits.

Another obvious criticism against our index is that its construction relies on the semi-external memory model, whereas all other indexes we compare against are constructed in internal memory. This is not an issue though as a practitioner would mostly care about how each implementation performs in practice. The fact that our construction relies on the semi-external memory model merely means that a reasonable amount of disk space should be utilized. To get rid of the semi-external memory model in our construction, it suffices to replace the construction of the suffix array in external memory with an efficient sparse suffix sorting algorithm; and this would (probably) only make our construction faster.

Finally, as mentioned in Section 5, although many other engineered implementations for classic text indexes exist, e.g. [39] for FM-index or [11] for suffix array, we have based our implementations on sdsl-lite as much as possible for fairness. Likewise, in the implementation of our index, we have used sdsl-lite as much as possible, and no particular low-level code optimization was conducted.

Future Work. Our immediate goal is to improve the construction time of the bd-anchors index in the following two ways. In our current implementation, we use the standard lexicographic order to compute minimizers. However, for an alphabet size σ, there are $\sigma!$ lexicographic orders that we could choose from. For large alphabets, like the one in SOURCES, a different lexicographic order could result in significantly fewer minimizers, and thus in a significantly faster bd-anchors computation. Second, as mentioned above, given the set of bd-anchors, we could replace the suffix array external-memory construction with an existing sparse suffix sorting algorithm.

ACKNOWLEDGMENTS

This research is supported in part: 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; and by UKRI through REPHRAIN (EP/V011189/1).

REFERENCES

[1] James Abello, Adam L. Buchsbaum, and Jeffery R. Westbrook. 2002. A Functional Approach to External Graph Algorithms. *Algorithmica* 32, 3 (2002), 437–458. https://doi.org/10.1007/s00453-001-0088-5

[2] Alberto Apostolico, Maxime Crochemore, Martin Farach-Colton, Zvi Galil, and S. Muthukrishnan. 2016. 40 years of suffix trees. *Commun. ACM* 59, 4 (2016), 66–73. https://doi.org/10.1145/2810036

[3] Mozhdeh Ariannezhad, Ali Montazeralghaem, Hamed Zamani, and Azadeh Shakery. 2017. 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*, Noriko Kando, Tetsuya Sakai, Hideo Joho, Hang Li, Arjen P. de Vries, and Ryen W. White (Eds.). ACM, 1201–1204. https://doi.org/10.1145/3077136.3080761

[4] Jérémy Barbay, Francisco Claude, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. 2014. Efficient Fully-Compressed Sequence Representations. *Algorithmica* 69, 1 (2014), 232–268. https://doi.org/10.1007/s00453-012-9726-3

[5] Djamal Belazzougui. 2014. 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*, David B. Shmoys (Ed.). ACM, 148–193. https://doi.org/10.1145/2591796.2591885

[6] Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. 2020. Linear-time String Indexing and Analysis in Small Space. *ACM Trans. Algorithms* 16, 2 (2020), 17:1–17:54. https://doi.org/10.1145/3381417

[7] Djamal Belazzougui and Gonzalo Navarro. 2015. Optimal Lower and Upper Bounds for Representing Sequences. *ACM Trans. Algorithms* 11, 4 (2015), 31:1–31:21. https://doi.org/10.1145/2629339

[8] Djamal Belazzougui and Simon J. Puglisi. 2016. 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*, Robert Krauthgamer (Ed.). SIAM, 2053–2071. https://doi.org/10.1137/1.9781611974331.ch143

[9] Stav Ben-Nun, Shay Golan, Tomasz Kociumaka, and Matan Kraus. 2020. 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)*, Inge Li Gørtz and Oren Weimann (Eds.), Vol. 161. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:14. https://doi.org/10.4230/LIPIcs.CPM.2020.5

[10] Michael Bendersky and W. Bruce Croft. 2008. 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*, Sung-Hyon Myaeng, Douglas W. Oard, Fabrizio Sebastiani, Tat-Seng Chua, and Mun-Kew Leong (Eds.). ACM, 491–498. https://doi.org/10.1145/1390334.1390419

[11] Nico Bertram, Jonas Ellert, and Johannes Fischer. 2021. Lyndon Words Accelerate Suffix Sorting, See [77], 15:1–15:13. https://doi.org/10.4230/LIPIcs.ESA.2021.15

[12] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. 2016. Inducing Suffix and LCP Arrays in External Memory. *ACM J. Exp. Algorithmics* 21, 1 (2016), 2.3:1–2.3:27. https://doi.org/10.1145/2975593

[13] Or Birenzwige, Shay Golan, and Ely Porat. 2020. 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*, Shuchi Chawla (Ed.). SIAM, 607–626. https://doi.org/10.1137/1.9781611975994.37

[14] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 11 (2020), 2649–2661. https://doi.org/10.14778/3407790.3407851

[15] Stefan Burkhardt and Juha Kärkkäinen. 2003. 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)*, Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore (Eds.), Vol. 2676. Springer, 55–69. https://doi.org/10.1007/3-540-44888-8_5

[16] Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. 2011. Orthogonal range searching on the RAM, revisited. In *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, Ferran Hurtado and Marc J. van Kreveld (Eds.). ACM, 1–10. https://doi.org/10.1145/1998196.1998198

[17] Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. 2018. 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)*, Gonzalo Navarro, David Sankoff, and Binhai Zhu (Eds.), Vol. 105. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:16. https://doi.org/10.4230/LIPIcs.CPM.2018.23

[18] Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. 2021. Faster Algorithms for Longest Common Substring, See [77], 30:1–30:17. https://doi.org/10.4230/LIPIcs.ESA.2021.30

[19] Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. 2022. Longest Palindromic Substring in Sublinear Time. In *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic (LIPIcs)*, Hideo Bannai and Jan Holub (Eds.), Vol. 223. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:9. https://doi.org/10.4230/LIPIcs.CPM.2022.20

[20] Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro (Eds.). 2015. *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings.* Lecture Notes in Computer Science, Vol. 9133. Springer. https://doi.org/10.1007/978-3-319-19929-0

[21] Francisco Claude, Gonzalo Navarro, Hannu Peltola, Leena Salmela, and Jorma Tarhio. 2012. String matching with alphabet sampling. *J. Discrete Algorithms* 11 (2012), 37–50. https://doi.org/10.1016/j.jda.2010.09.004

[22] Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. 2015. Suffix Trays and Suffix Trists: Structures for Faster Text Indexing. *Algorithmica* 72, 2 (2015), 450–466. https://doi.org/10.1007/s00453-013-9860-6

[23] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. 2007. *Algorithms on strings.* Cambridge University Press.

[24] Patrick Dinklage, Johannes Fischer, and Alexander Herlez. 2021. Engineering Predecessor Data Structures for Dynamic Integer Sets. In *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France (LIPIcs)*, David Coudert and Emanuele Natale (Eds.), Vol. 190. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:19. https://doi.org/10.4230/LIPIcs.SEA.2021.7

[25] Patrick Dinklage, Johannes Fischer, Alexander Herlez, Tomasz Kociumaka, and Florian Kurpicz. 2020. Practical Performance of Space Efficient Data Structures for Longest Common Extensions, See [43], 39:1–39:20. https://doi.org/10.4230/LIPIcs.ESA.2020.39

[26] Martin Farach. 1997. 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.* IEEE Computer Society, 137–143. https://doi.org/10.1109/SFCS.1997.646102

[27] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. 2008. Compressed text indexes: From theory to practice. *ACM J. Exp. Algorithmics* 13 (2008). https://doi.org/10.1145/1412228.1455268

[28] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (2005), 552–581. https://doi.org/10.1145/1082036.1082039

[29] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. 2004. 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)*, Alberto Apostolico and Massimo Melucci (Eds.), Vol. 3246. Springer, 150–160. https://doi.org/10.1007/978-3-540-30213-1_23

[30] Johannes Fischer and Pawel Gawrychowski. 2015. Alphabet-Dependent String Searching with Wexponential Search Trees, See [20], 160–171. https://doi.org/10.1007/978-3-319-19929-0_14

[31] Johannes Fischer and Volker Heun. 2011. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM J. Comput.* 40, 2 (2011), 465–492. https://doi.org/10.1137/090779759

[32] Gianni Franceschini and S. Muthukrishnan. 2007. 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)*, Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki (Eds.), Vol. 4596. Springer, 533–545. https://doi.org/10.1007/978-3-540-73420-8_47

[33] Michael L. Fredman, János Komlós, and Endre Szemerédi. 1984. Storing a Sparse Table with 0(1) Worst Case Access Time. *J. ACM* 31, 3 (1984), 538–544. https://doi.org/10.1145/828.1884

[34] Michael L. Fredman and Dan E. Willard. 1990. 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*, Harriet Ortiz (Ed.). ACM, 1–7. https://doi.org/10.1145/100216.100217

[35] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. 2020. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *J. ACM* 67, 1 (2020), 2:1–2:54. https://doi.org/10.1145/3375890

[36] Younan Gao, Meng He, and Yakov Nekrich. 2020. Fast Preprocessing for Optimal Orthogonal Range Reporting and Range Successor with Applications to Text Indexing, See [43], 54:1–54:18. https://doi.org/10.4230/LIPIcs.ESA.2020.54

[37] Pawel Gawrychowski and Tomasz Kociumaka. 2017. Sparse Suffix Tree Construction in Optimal Time and Space. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, Philip N. Klein (Ed.). SIAM, 425–439. https://doi.org/10.1137/1.9781611974782.27

[38] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. 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)*, Joachim Gudmundsson and Jyrki Katajainen (Eds.), Vol. 8504. Springer, 326–337. https://doi.org/10.1007/978-

3-319-07959-2_28

[39] Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. 2019. Fixed Block Compression Boosting in FM-Indexes: Theory and Practice. *Algorithmica* 81, 4 (2019), 1370–1391. https://doi.org/10.1007/s00453-018-0475-9

[40] Simon Gog, Alistair Moffat, and Matthias Petri. 2017. 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*, Sándor P. Fekete and Vijaya Ramachandran (Eds.). SIAM, 73–82. https://doi.org/10.1137/1.9781611974768.6

[41] Keisuke Goto. 2019. 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*, Jan Holub and Jan Zdárek (Eds.). Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 111–125. http://www.stringology.org/event/2019/p11.html

[42] Szymon Grabowski and Marcin Raniszewski. 2017. Sampled suffix array with minimizers. *Softw. Pract. Exp.* 47, 11 (2017), 1755–1771. https://doi.org/10.1002/spe.2481

[43] Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders (Eds.). 2020. *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference).* LIPIcs, Vol. 173. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://www.dagstuhl.de/dagpub/978-3-95977-162-7

[44] Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.* 35, 2 (2005), 378–407. https://doi.org/10.1137/S0097539702402354

[45] Manish Gupta and Michael Bendersky. 2015. 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*, Ricardo Baeza-Yates, Mounia Lalmas, Alistair Moffat, and Berthier A. Ribeiro-Neto (Eds.). ACM, 1121–1124. https://doi.org/10.1145/2766462.2767877

[46] Dan Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press. https://doi.org/10.1017/cbo9780511574931

[47] Monika Rauch Henzinger. 2006. 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*, Efthimis N. Efthimiadis, Susan T. Dumais, David Hawking, and Kalervo Järvelin (Eds.). ACM, 284–291. https://doi.org/10.1145/1148170.1148222

[48] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. 2009. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. *SIAM J. Comput.* 38, 6 (2009), 2162–2178. https://doi.org/10.1137/070685373

[49] Tomohiro I, Juha Kärkkäinen, and Dominik Kempa. 2014. Faster Sparse Suffix Sorting. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France (LIPIcs)*, Ernst W. Mayr and Natacha Portier (Eds.), Vol. 25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 386–396. https://doi.org/10.4230/LIPIcs.STACS.2014.386

[50] Chirag Jain, Arang Rhie, Nancy Hansen, Sergey Koren, and Adam M. Phillippy. 2022. Long-read mapping to repetitive reference sequences using Winnowmap2. *Nat Methods* 19 (2022), 705–710. https://doi.org/10.1038/s41592-022-01457-8

[51] Jiaojiao Jiang, Steve Versteeg, Jun Han, Md. Arafat Hossain, Jean-Guy Schneider, Christopher Leckie, and Zeinab Farahmandpour. 2019. P-Gram: Positional N-Gram for the Clustering of Machine-Generated Messages. *IEEE Access* 7 (2019), 88504–88516. https://doi.org/10.1109/ACCESS.2019.2924928

[52] Juha Kärkkäinen and Dominik Kempa. 2016. LCP Array Construction in External Memory. *ACM J. Exp. Algorithmics* 21, 1 (2016), 1.7:1–1.7:22. https://doi.org/10.1145/2851491

[53] Juha Kärkkäinen and Dominik Kempa. 2016. LCP Array Construction Using O(sort(n)) (or Less) I/Os. In *String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings (Lecture Notes in Computer Science)*, Shunsuke Inenaga, Kunihiko Sadakane, and Tetsuya Sakai (Eds.), Vol. 9954. 204–217. https://doi.org/10.1007/978-3-319-46049-9_20

[54] Juha Kärkkäinen and Dominik Kempa. 2019. Better External Memory LCP Array Construction. *ACM J. Exp. Algorithmics* 24, 1 (2019), 1.3:1–1.3:27. https://doi.org/10.1145/3297723

[55] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. 2015. Parallel External Memory Suffix Sorting, See [20], 329–342. https://doi.org/10.1007/978-3-319-19929-0_28

[56] Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi, and Bella Zhukova. 2017. 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*, Sándor P. Fekete and Vijaya Ramachandran (Eds.). SIAM, 98–108. https://doi.org/10.1137/1.9781611974768.8

[57] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. 2006. Linear work suffix array construction. *J. ACM* 53, 6 (2006), 918–936. https://doi.org/10.1145/1217856.1217858

[58] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. 2001. 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)*, Amihood Amir and Gad M. Landau (Eds.), Vol. 2089. Springer, 181–192. https://doi.org/10.1007/3-540-48194-X_17

[59] Dominik Kempa and Tomasz Kociumaka. 2019. 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*, Moses Charikar and Edith Cohen (Eds.). ACM, 756–767. https://doi.org/10.1145/3313276.3316368

[60] Dominik Kempa and Tomasz Kociumaka. 2023. 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*, Nikhil Bansal and Viswanath Nagarajan (Eds.). SIAM, 5122–5202. https://doi.org/10.1137/1.9781611977554.ch187

[61] Tomasz Kociumaka. 2016. 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)*, Roberto Grossi and Moshe Lewenstein (Eds.), Vol. 54. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:12. https://doi.org/10.4230/LIPIcs.CPM.2016.28

[62] Stefan Kurtz. 1999. Reducing the space requirement of suffix trees. *Softw. Pract. Exp.* 29, 13 (1999), 1149–1171. https://doi.org/10.1002/(SICI)1097-024X(199911)29:13%3C1149::AID-SPE274%3E3.0.CO;2-O

[63] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. 2009. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology* 10, 3 (2009), R25. https://doi.org/10.1186/gb-2009-10-3-r25

[64] Heng Li and Richard Durbin. 2009. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinform.* 25, 14 (2009), 1754–1760. https://doi.org/10.1093/bioinformatics/btp324

[65] Ruiqiang Li, Chang Yu, Yingrui Li, Tak Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. 2009. SOAP2: an improved ultrafast tool for short read alignment. *Bioinform.* 25, 15 (2009), 1966–1967. https://doi.org/10.1093/bioinformatics/btp336

[66] Zhize Li, Jian Li, and Hongwei Huo. 2022. Optimal in-place suffix sorting. *Inf. Comput.* 285, Part (2022), 104818. https://doi.org/10.1016/j.ic.2021.104818

[67] Glennis A. Logsdon, Mitchell R. Vollger, and Evan E. Eichler. 2020. Long-read human genome sequencing and its applications. *Nat. Rev. Genet.* 21, 10 (2020), 597–614. https://doi.org/10.1038/s41576-020-0236-x

[68] Grigorios Loukides and Solon P. Pissis. 2021. Bidirectional String Anchors: A New String Sampling Mechanism, See [77], 64:1–64:21. https://doi.org/10.4230/LIPIcs.ESA.2021.64

[69] Grigorios Loukides, Solon P. Pissis, and Michelle Sweering. 2023. Bidirectional String Anchors for Improved Text Indexing and Top-K Similarity Search. *IEEE Trans. Knowl. Data Eng.* (2023). https://doi.org/10.1109/TKDE.2022.3231780

[70] Mamoru Maekawa. 1985. A Square Root N Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Trans. Comput. Syst.* 3, 2 (1985), 145–159.

[71] Veli Mäkinen and Gonzalo Navarro. 2006. 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)*, José R. Correa, Alejandro Hevia, and Marcos A. Kiwi (Eds.), Vol. 3887. Springer, 703–714. https://doi.org/10.1007/11682462_64

[72] Udi Manber and Eugene W. Myers. 1993. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.* 22, 5 (1993), 935–948. https://doi.org/10.1137/0222058

[73] Olena Medelyan and Ian H. Witten. 2006. Thesaurus based automatic keyphrase indexing. In *ACM/IEEE Joint Conference on Digital Libraries, JCDL 2006, Chapel Hill, NC, USA, June 11-15, 2006, Proceedings*, Gary Marchionini, Michael L. Nelson, and Catherine C. Marshall (Eds.). ACM, 296–297. https://doi.org/10.1145/1141753.1141819

[74] Donald R. Morrison. 1968. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (1968), 514–534. https://doi.org/10.1145/321479.321481

[75] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. 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*, Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy (Eds.). OpenProceedings.org, 283–294. https://doi.org/10.5441/002/edbt.2014.27

[76] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. 2017. 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.* 408–424. https://doi.org/10.1137/1.9781611974782.26

[77] Petra Mutzel, Rasmus Pagh, and Grzegorz Herman (Eds.). 2021. *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference).* LIPIcs, Vol. 204. Schloss Dagstuhl - Leibniz-Zentrum für

Informatik. https://www.dagstuhl.de/dagpub/978-3-95977-204-4

[78] Gonzalo Navarro. 2016. *Compact Data Structures - A Practical Approach.* Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/compact-data-structures-practical-approach?format=HB

[79] Gonzalo Navarro and Yakov Nekrich. 2017. Time-Optimal Top-k Document Retrieval. *SIAM J. Comput.* 46, 1 (2017), 80–113. https://doi.org/10.1137/140998949

[80] Enno Ohlebusch, Johannes Fischer, and Simon Gog. 2010. 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)*, Edgar Chávez and Stefano Lonardi (Eds.), Vol. 6393. Springer, 322–333. https://doi.org/10.1007/978-3-642-16321-0_34

[81] Nicola Prezza. 2021. Optimal Substring Equality Queries with Applications to Sparse Text Indexing. *ACM Trans. Algorithms* 17, 1 (2021), 7:1–7:23. https://doi.org/10.1145/3426870

[82] Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. 2004. Reducing storage requirements for biological sequence comparison. *Bioinform.* 20, 18 (2004), 3363–3369. https://doi.org/10.1093/bioinformatics/bth408

[83] Patricia Rodriguez-Tomé, Peter Stoehr, Graham Cameron, and Tomas P. Flores. 1996. The European Bioinformatics Institute (EBI) databases. *Nucleic Acids Res.* 24, 1 (1996), 6–12. https://doi.org/10.1093/nar/24.1.6

[84] Kunihiko Sadakane. 2007. Compressed Suffix Trees with Full Functionality. *Theory Comput. Syst.* 41, 4 (2007), 589–607. https://doi.org/10.1007/s00224-006-1198-x

[85] Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. 2003. 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*, Alon Y. Halevy, Zachary G. Ives, and AnHai Doan (Eds.). ACM, 76–85. https://doi.org/10.1145/872757.872770

[86] Kazutoshi Umemoto, Ruihua Song, Jian-Yun Nie, Xing Xie, Katsumi Tanaka, and Yong Rui. 2017. Search by Screenshots for Universal Article Clipping in Mobile Apps. *ACM Trans. Inf. Syst.* 35, 4 (2017), 34:1–34:29. https://doi.org/10.1145/3091107

[87] Jeffrey Scott Vitter. 2006. Algorithms and Data Structures for External Memory. *Found. Trends Theor. Comput. Sci.* 2, 4 (2006), 305–474. https://doi.org/10.1561/0400000014

[88] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. 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*, Alexander Böhm and Tilmann Rabl (Eds.). ACM, 1:1–1:6. https://doi.org/10.1145/3209950.3209952

[89] Peter Weiner. 1973. Linear Pattern Matching Algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*. IEEE Computer Society, 1–11. https://doi.org/10.1109/SWAT.1973.13

[90] Aaron M. Wenger et al. 2019. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nat. Biotechnol.* 37 (2019), 1155–1162. https://doi.org/10.1038/s41587-019-0217-9

[91] Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. 2020. Improved design and analysis of practical minimizers. *Bioinform.* 36, Supplement-1 (2020), i119–i127. https://doi.org/10.1093/bioinformatics/btaa472