DOI:
10.1109/TKDE.2022.3231780

Document Version
Peer reviewed version

Bidirectional String Anchors for Improved Text Indexing and Top-K Similarity Search

Grigorios Loukides *Senior Member, IEEE*, Solon P. Pissis, and Michelle Sweering

Abstract—The minimizers sampling mechanism is a popular mechanism for string sampling. However, minimizers sampling mechanisms lack good guarantees on the expected size of their samples for different combinations of their input parameters. Furthermore, indexes constructed over minimizers samples lack good worst-case guarantees for on-line pattern searches. In response, we propose bidirectional string anchors (bd-anchors), a new string sampling mechanism. Given an integer ℓ, our mechanism selects the lexicographically smallest rotation in every length-ℓ fragment. We show that, like minimizers samples, bd-anchors samples are approximately uniform, locally consistent, and computable in linear time. Furthermore, our experiments demonstrate that the bd-anchors sample sizes decrease proportionally to ℓ; and that these sizes are *competitive to or smaller than* the minimizers sample sizes. We theoretically justify these results by analyzing the expected size of bd-anchors samples. We also prove that computing a total order on the input alphabet which minimizes the bd-anchors sample size is NP-hard. We next highlight the benefits of bd-anchors in two important applications: text indexing and top-K similarity search. For the first application, we develop an index for performing on-line pattern searches in near-optimal time, and show experimentally that a simple implementation of our index is *consistently faster* for on-line pattern searches than an analogous implementation of a minimizers-based index; we also show that it is *substantially* faster than two classic text indexes. For the second application, we develop a heuristic for top-K similarity search under edit distance, and show experimentally that it is generally as *accurate* as the state-of-the-art tool for the same purpose but *more than one order of magnitude faster*.

Index Terms—string algorithms, string sampling, text indexing, top-K string similarity search

1 Introduction

A large number of information systems are fueled by string (sequential) data. A string T is a sequence of letters over some alphabet Σ. For example, when Σ is a set of items, T can represent a user's purchasing history [1]; when Σ is a set of locations, T can represent a user's location profile [82]; and when T is the set of DNA bases, T can represent a genome sequence [51]. With the increasing size of string datasets that require processing, traditional string-processing tools, like suffix trees [79] or suffix arrays [62], have become prohibitive to use. These traditional tools are thus progressively being replaced or adapted by string sampling mechanisms [73], [71], [28], [37], [50], [33]. Such mechanisms can be fundamentally different from each other depending on their objectives or the intended application.

In this paper, we focus on *minimizers*, a popular mechanism for string sampling (also known as *winnowing*), which was introduced independently by Schleimer et al. [73] and by Roberts et al. [71]. The minimizers mechanism samples a set of positions over an input string. In particular, the goal of this sampling mechanism is, given a string T of length n over an alphabet Σ of size σ, to simultaneously satisfy the following properties:

- *Property 1 (approximately uniform sampling):* Every sufficiently long fragment of T has a representative position sampled by the mechanism. This ensures that a sufficiently long pattern will not be missed during search.
- *Property 2 (local consistency):* Exact matches between sufficiently long fragments of T are preserved unconditionally by having the same (relative) representative positions sampled by the mechanism. This ensures that similarity between sufficiently long similar strings will be preserved during search.

In most practical scenarios, sampling the smallest number of positions is desirable, as long as Properties 1 and 2 are satisfied. This is because it leads to small data structures or fewer computations. Indeed, the minimizers sampling mechanism satisfies the property of approximately uniform sampling: given two positive integers w and k, it selects at least one length-k substring in every fragment of w consecutive length-k substrings (Property 1). Specifically, this is achieved by selecting the starting positions of the smallest length-k substrings in every $(w+k-1)$-long fragment, where smallest is defined by a choice of a total order on the universe of length-k strings. These positions are called the "minimizers". Thus from similar fragments, similar length-k substrings are sampled (Property 2). In particular, if two strings have a fragment of length $w+k-1$ in common, then they have at least one minimizer corresponding to the same length-k substring. Let us denote by $\mathcal{M}_{w,k}(T)$ the set of minimizers of string T. The following example illustrates the sampling.

Example 1. The set $\mathcal{M}_{w,k}$ of minimizers for $w=k=3$ for string $T=$ `aabaaabcbda` (using a 1-based index) is $\mathcal{M}_{3,3}(T)=\{1,4,5,6,7\}$ and for string $Q=$ `abaaa` is $\mathcal{M}_{3,3}(Q)=\{3\}$. Indeed Q occurs at position 2 in T; and

- *G. Loukides is with King's College London, United Kingdom. Email: grigorios.loukides@kcl.ac.uk*
- *S. P. Pissis is with CWI and the Vrije Universiteit, The Netherlands. Email: solon.pissis@cwi.nl*
- *M. Sweering is with CWI, The Netherlands. Email:michelle.sweering@cwi.nl*

Manuscript received...

Q and $T[2\,..\,6]$ have the minimizers 3 and 4, respectively, which both correspond to string `aaa` of length $k = 3$.

The minimizers sampling mechanism is very versatile, and it has been employed in various ways in many different applications [57], [80], [22], [14], [36], [42], [43], [58], [44], [67], [31]. Since its inception, the minimizers sampling mechanism has undergone numerous theoretical and practical improvements [68], [14], [65], [64], [18], [26], [86], [44], [89], [25], [39] with a particular focus on minimizing the size of the residual sample; see Section 6 for a summary on this line of research.

1.1 Our Motivation

Although minimizers have been extensively and successfully used, especially in bioinformatics, we observe several inherent problems with setting the parameters w and k. In particular, although the notion of length-k substrings (known as *k-mers* or *k-grams*) is a widely-used string processing tool, we argue that, in the context of minimizers, it may be causing many more problems than it solves: it is not clear to us why one should use an extra sampling parameter k to effectively characterize a fragment of length $\ell = w + k - 1$ of T. In what follows, we describe some problems that may arise when setting the parameters w and k.

- *Indexing:* The most widely-used approach is to index the selected minimizers using a hash table. The *key* is the selected length-k substring and the *value* is the list of positions it occurs. If one would like to use length-k' substrings for the minimizers with $\ell = w + k - 1 = w' + k' - 1$, for some $w' \neq w$ and $k' \neq k$, they should compute the new set $\mathcal{M}_{w',k'}(T)$ of minimizers and construct their new index based on $\mathcal{M}_{w',k'}$ from scratch.
- *Querying:* To the best of our knowledge, no index based on minimizers can return in optimal or near-optimal time all occurrences of a pattern Q of length $|Q| \geq \ell = w + k - 1$ in T.
- *Sample Size:* If one would like to minimize the number of selected minimizers, they should consider different total orders on the universe of length-k strings, which may complicate practical implementations, often scaling only up to a small k value, e.g. $k = 16$ [26]. On the other hand, when k is fixed and w increases, the length-k substrings in a fragment become increasingly decoupled from each other, and that *regardless of the total order* we may choose. Unfortunately, this interplay phenomenon is inherent in minimizers. It is known that $k \geq \log_\sigma(w) + c$, for a fixed constant c, is a *necessary condition* for the existence of minimizers samples with expected size in $\mathcal{O}(n/w)$ [86]; see Section 6.

We propose the notion of bidirectional string anchors (bd-anchors) to alleviate these disadvantages. The bd-anchors is a mechanism that drops the sampling parameter k and its corresponding disadvantages. We only fix a parameter ℓ, which can be viewed as the length $w+k-1$ of the fragments in the minimizers sampling mechanism. The *bd-anchor* of a string X of length ℓ is the lexicographically smallest rotation (cyclic shift) of X. We unambiguously characterize this rotation by its leftmost starting position in string XX. The set $\mathcal{A}_\ell(T)$ of the order-ℓ bd-anchors of string T is the set of bd-anchors of all length-ℓ fragments of T. It can be readily verified that bd-anchors satisfy Properties 1 and 2.

Example 2. The set $\mathcal{A}_\ell(T)$ of bd-anchors for $\ell = 5$ for string $T =$ `aabaaabcbda` (using a 1-based index) is $\mathcal{A}_5(T) = \{4, 5, 6, 11\}$ and for string $Q =$ `abaaa`, $\mathcal{A}_5(Q) = \{3\}$. Indeed, Q occurs at position 2 in T; and Q and $T[2\,..\,6]$ have the bd-anchors 3 and 4, respectively, which both correspond to the rotation `aaaab`.

Let us remark that *string synchronizing* sets, recently introduced by Kempa and Kociumaka [50], is a string sampling mechanism which may also be employed to resolve the disadvantages of minimizers. Yet, it appears to be quite complicated to be efficient in practice. For instance, in [23], the authors used a simplified and specific definition of string synchronizing sets to design a space-efficient data structure for answering longest common extension queries.

1.2 Our Contributions

We provide theoretical and experimental justification of the benefits of bd-anchors. In particular, we show these benefits in the context of two important string-processing tasks: text indexing and top-K similarity search. We consider the word RAM model of computations with w-bit machine words, where $w = \Omega(\log n)$, for stating our results. We also assume throughout that string T is over alphabet $\Sigma = \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$, which captures virtually any real-world scenario. We measure space in terms of w-bit machine words. We next provide a summary of our contributions:

1) *Construction and Sample Size (Section 3):* First, we provide upper bounds on the time and space required to construct the set $\mathcal{A}_\ell(T)$. We show that $\mathcal{A}_\ell(T)$, for any $\ell > 0$ and any T of length n, can be constructed in $\mathcal{O}(n)$ time. We generalize this result showing that for any constant $\epsilon \in (0, 1]$, $\mathcal{A}_\ell(T)$ can be constructed in $\mathcal{O}(n + n^{1-\epsilon}\ell)$ time using $\mathcal{O}(n^\epsilon + \ell + |\mathcal{A}_\ell|)$ space. Second, we show that the expected size of $\mathcal{A}_\ell$ for strings of length n, randomly generated by a memoryless source with identical letter probabilities, is in $\mathcal{O}(n/\ell)$, for any integer $\ell > 0$. The latter is in contrast to minimizers which achieve the expected bound of $\mathcal{O}(n/w)$ only when $k \geq \log_\sigma w + c$, for some constant c [86]. Third, we show a negative result using a reduction from minimum feedback arc set: computing a total order $\leq$ on Σ which minimizes $|\mathcal{A}_\ell(T)|$ is NP-hard. Finally, we compare the size of bd-anchors samples to the size of minimizers samples. We show, using five real datasets, that indeed the size of $\mathcal{A}_\ell$ decreases proportionally to ℓ; that it is *competitive to or smaller than* $\mathcal{M}_{w,k}$, when $\ell = w + k - 1$; and that it is *much smaller* than $\mathcal{M}_{w,k}$ for *small* w values, which is practically important, as widely-used aligners that are based on minimizers will require less space and computation time if bd-anchors are used instead.
2) *Text Indexing (Section 4):* We start by showing how bd-anchors can be used to construct an efficient index for on-line pattern searches. Text indexing is a fundamental and extensively studied problem [79], [62], [27], [46], [28], [37], [40], [7], [16], [66], [50], [33] with numerous applications in information retrieval [5] and bioinformatics [38].

We show an index based on $\mathcal{A}_\ell(T)$, for any string T of length n and any integer $\ell > 0$, which answers on-line pattern searches in near-optimal time. In particular, for any constant $\epsilon > 0$, we show that our index supports the following space/query-time trade-offs:

- it occupies $\mathcal{O}(|\mathcal{A}_\ell(T)|)$ extra space and reports all k occurrences of any pattern Q of length $|Q| \geq \ell$ given on-line in $\mathcal{O}(|Q| + (k+1)\log^\epsilon(|\mathcal{A}_\ell(T)|))$ time; or
- it occupies $\mathcal{O}(|\mathcal{A}_\ell(T)|\log^\epsilon(|\mathcal{A}_\ell(T)|))$ extra space and reports all k occurrences of any pattern Q of length $|Q| \geq \ell$ given on-line in $\mathcal{O}(|Q| + \log\log(|\mathcal{A}_\ell(T)|) + k)$ time.

We also show that our index can be constructed in $\mathcal{O}(n+|\mathcal{A}_\ell(T)|\sqrt{\log(|\mathcal{A}_\ell(T)|)})$ time. We then show, using five real datasets, that a simple implementation of our index is *consistently faster* in on-line pattern searches than an analogous implementation of the minimizers-based index proposed by Grabowski and Raniszewski in [36]. Notably, we also show that our index is *substantially faster* than two classic text indexes [62], [28] that are commonly employed in widely-used software [54], [59].

3) *Top-K Similarity Search (Section 5):* We highlight the applicability of bd-anchors by developing an efficient and effective heuristic for top-K similarity search under edit distance. This is also a fundamental and extensively studied problem [45], [13], [15], [56], [81], [85], [69], [78], [77], [20], [41], [83], [84] with numerous applications in databases and data mining [70] and in other areas including information retrieval [5] and bioinformatics [53]. We show, using both synthetic and real datasets, that while our heuristic is generally as *accurate* as the state-of-the-art tool for top-K similarity searches, proposed by Zhang and Zhang [84], it is *more than one order of magnitude faster*.

In Section 2, we provide some necessary preliminaries; and, in Section 6, we discuss related work with a special focus on minimizers. Let us stress that, although other works may be related to our contributions, we focus on comparing mostly to minimizers because these are extensively used in applications. Section 7 concludes the paper.

A preliminary version of this paper was announced at ESA 2021 [60]. The current version contains the following additional material: (1) full proofs regarding the construction of bd-anchors; (2) a new algorithm allowing to compute bd-anchors efficiently using (strongly) sublinear space; (3) a theoretical analysis of the expected size of bd-anchors; (4) a new version of bd-anchors, referred to as "reduced bd-anchors"; (5) a non-trivial NP-hardness proof for finding a total order on the input alphabet that minimizes the number of bd-anchors; (6) a new algorithm allowing to query multiple fragments; and (7) additional experimental results.

1.3 Our Techniques

We begin by providing an informal overview of our techniques in Figure 1. In order to arrive at these results several technical *challenges* had to be addressed. We give a brief description of each challenge next:

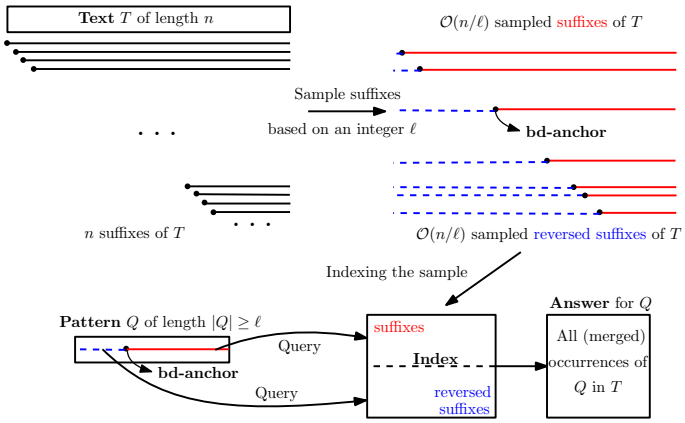


Fig. 1: An informal overview of our techniques: Given a text T of length n, we consider all of its n positions and sample them for a given integer ℓ using the bd-anchors mechanism. The selected positions (bd-anchors) imply a set of $\mathcal{O}(n/\ell)$ suffixes and a set of $\mathcal{O}(n/\ell)$ reversed suffixes of T. We then index these strings, so as to answer pattern matching queries for patterns of length at least ℓ. When such a pattern Q is given, we find any of its bd-anchors, which implies a suffix and a reversed suffix of Q, and query those using our index. Our index efficiently merges the partial results (i.e., occurrences of suffixes and reversed suffixes), and returns all occurrences of Q in T in nearly-optimal time.

- *Sample Size:* Proving the existence of minimizers samples with expected size in $\mathcal{O}(n/w)$ is relatively simple (see the proofs in [73] and [71]). The additional technical difficulty with bd-anchors comes from the fact that bd-anchors "wrap-around" and so the rotations always change as we move from one window to another. By analyzing and treating separately the cases $\log\ell \leq \log\sigma$ and $\log\ell > \log\sigma$, we show that the expected size of bd-anchors samples is in $\mathcal{O}(n/\ell)$.
- *Constructing the Sample*: Constructing minimizers samples in linear time is also relatively simple using hashing (see [73]). Again, the additional technical difficulty with bd-anchors comes from the fact that rotations always change as we move from one window to another, and so it is not trivial to find lexicographically smallest rotations in $\mathcal{O}(1)$ time per window. To achieve this, we employ the data structure of Kociumaka [52] and some combinatorial observations.
- *Indexing with Guarantees*: Grabowski and Raniszewski [36] show how minimizers samples can be used to improve text indexing both in space and query time. Although their approach shows improved query times for on-line pattern searches in practice, it lacks good worst-case guarantees. To achieve near-optimal time for on-line pattern searches we exploit a connection between string algorithms and computational geometry (2D range reporting) [3].

2 Preliminaries

We start with some basic definitions and notation following [17]. An *alphabet* Σ is a finite nonempty set of elements called *letters*. A *string* $X = X[1]\dots X[n]$ is a sequence of

length $|X| = n$ of letters from Σ. The *empty* string, denoted by ε, is the string of length 0. The fragment $X[i\,..\,j]$ of X is an *occurrence* of the underlying *substring* $S = X[i]\dots X[j]$. We also say that S occurs at *position* i in X. A *prefix* of X is a fragment of X of the form $X[1\,..\,j]$ and a *suffix* of X is a fragment of X of the form $X[i\,..\,n]$. The set of all strings over Σ (including ε) is denoted by Σ^*. The set of all length-k strings over Σ is denoted by Σ^k. Given two strings X and Y, the *edit distance* $d_{\mathrm{E}}(X, Y)$ is the minimum number of edit operations (letter insertion, deletion, or substitution) transforming one string into the other.

Let M be a finite nonempty set of strings over Σ of total length m. We define the *trie* of M, denoted by $\mathsf{TR}(M)$, as a deterministic finite automaton that recognizes M. Its set of states (nodes) is the set of prefixes of the elements of M; the initial state (root node) is ε; the set of terminal states (leaf nodes) is M; and edges are of the form $(u, \alpha, u\alpha)$, where u and $u\alpha$ are nodes and $\alpha \in \Sigma$. The size of $\mathsf{TR}(M)$ is thus $\mathcal{O}(m)$. The *compacted trie* of M, denoted by $\mathsf{CT}(M)$, contains the root node, the branching nodes, and the leaf nodes of $\mathsf{TR}(M)$. The term compacted refers to the fact that $\mathsf{CT}(M)$ reduces the number of nodes by replacing each maximal branchless path segment with a single edge, and that it uses a fragment of a string $s \in M$ to represent the label of this edge in $\mathcal{O}(1)$ machine words. The size of $\mathsf{CT}(M)$ is thus $\mathcal{O}(|M|)$. When M is the set of suffixes of a string Y, then $\mathsf{CT}(M)$ is called the *suffix tree* of Y, and we denote it by $\mathsf{ST}(Y)$. The *suffix array* of Y is the lexicographically sorted array of the set of suffixes of Y, and we denote it by $\mathsf{SA}(Y)$. (We will often drop (Y) when it is clear from the context.) The suffix tree and the suffix array of a string of length n over an alphabet $\Sigma = \{1, \dots, n^{\mathcal{O}(1)}\}$ can be constructed in $\mathcal{O}(n)$ time [27]. We next present an example of these structures.

Example 3. Let $Y =$ `CAGAGA$` and assume that `$` is a terminal letter, which is the lexicographically smallest letter occurring in Y. $\mathsf{SA}(Y)$ is on the left and $\mathsf{ST}(Y)$ on the right.

i	$\mathsf{SA}[i]$	Suffix
1	7	`$`
2	6	`A$`
3	4	`AGA$`
4	2	`AGAGA$`
5	1	`CAGAGA$`
6	5	`GA$`
7	3	`GAGA$`

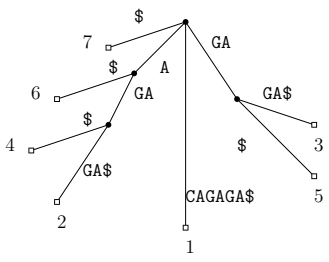


Fig. 2: $\mathsf{SA}(Y)$ and $\mathsf{SF}(Y)$.

Let us fix throughout a string $T = T[1\,..\,n]$ of length $|T| = n$ over an ordered alphabet Σ. Recall that we make the standard assumption of an integer alphabet $\Sigma = \{1, 2, \dots, n^{\mathcal{O}(1)}\}$.

We start by defining the notion of minimizers of T from [71] (the definition in [73] is slightly different). Given an integer $k > 0$, an integer $w > 0$, and the ith length-$(w+k-1)$ fragment $F = T[i\,..\,i+w+k-2]$ of T, we define the (w, k)-*minimizers* of F as the positions $j \in [i, i+w)$ where a lexicographically minimal length-k substring of F occurs. The set $\mathcal{M}_{w,k}(T)$ of (w, k)-minimizers of T is defined as the set of (w, k)-minimizers of $T[i\,..\,i+w+k-2]$, for all $i \in [1, n-w-k+2]$. The *density* of $\mathcal{M}_{w,k}(T)$ is defined as the quantity $|\mathcal{M}_{w,k}(T)|/n$. The following bounds are obtained trivially. The density of any minimizer scheme is at least $1/w$, since at least one (w, k)-minimizer is selected in each fragment, and at most 1, when every (w, k)-minimizer is selected.

If we waive the lexicographic order assumption, the set $\mathcal{M}_{w,k}(T)$ can be computed on-line in $\mathcal{O}(n)$ time, and if we further assume a constant-time computable function that gives us an *arbitrary* rank for each length-k substring in Σ^k in constant amortized time [44]. This can be implemented, for instance, using a rolling hash function (e.g. Karp-Rabin fingerprints [48]), and the rank (total order) is defined by this function. We also provide here, for completeness, a simple off-line $\mathcal{O}(n)$-time algorithm that uses a lexicographic order.

Theorem 1. *The set $\mathcal{M}_{w,k}(T)$, for any integers $w, k > 0$ and any string T of length n, can be constructed in $\mathcal{O}(n)$ time.*

Proof. The underlying algorithm has two main steps. In the first step, we construct $\mathsf{ST}(T)$, the suffix tree of T in $\mathcal{O}(n)$ time [27]. Using a depth-first search traversal of $\mathsf{ST}(T)$ we assign at every position of T in $[1, n-k+1]$ the lexicographic rank of $T[i\,..\,i+k-1]$ among all the length-k strings occurring in T. This process clearly takes $\mathcal{O}(n)$ time as $\mathsf{ST}(T)$ is an ordered structure; it yields an array R of size $n-k+1$ with lexicographic ranks. In the second step, we apply a folklore algorithm, which computes the minimum elements in a sliding window of size w (cf. [44]) over R. The set of reported indices is $\mathcal{M}_{w,k}(T)$. □

3 Bidirectional String Anchors

In Section 3.1, we examine the construction and the expected size of bd-anchors samples from a theoretical perspective, providing upper bounds as well as a non-trivial hardness result. In Section 3.2, we experimentally evaluate the size of bd-anchors samples and compare it to the respective size of minimizers samples.

3.1 Construction and Sample Size

We introduce the notion of bidirectional string anchors (bd-anchors). Given a string W, a string R is a *rotation* (or cyclic shift or conjugate) of W if and only if there exists a decomposition $W = UV$ such that $R = VU$, for a string U and a nonempty string V. We often characterize R by its starting position $|U| + 1$ in $WW = UVUV$. We use the term rotation interchangeably to refer to string R or to its identifier $(|U| + 1)$.

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

The *density* of $\mathcal{A}_\ell(T)$ is defined as the quantity $|\mathcal{A}_\ell(T)|/n$. It can be readily verified that the bd-anchors sampling mechanism satisfies Properties 1 (approximately uniform sampling) and 2 (local consistency).

Example 4. Let $\ell = 5$, $T = \texttt{aabaaabcbda}$, and $T' = \texttt{aacaaaccbda}$. Strings T and T' (are at Hamming distance 2 but) have the same set of bd-anchors of order 5: $\mathcal{A}_5(T) = \mathcal{A}_5(T') = \{4, 5, 6, 11\}$. The reader can probably share the intuition that the bd-anchors sampling mechanism is suitable for sequence comparison due to Properties 1 and 2, in particular, when the parameter ℓ is set accordingly.

3.1.1 Linear-Time Construction of $\mathcal{A}_\ell$

Importantly, we show that $\mathcal{A}_\ell$ admits an efficient construction. One can use the linear-time algorithm by Booth [10] to compute the lexicographically minimal rotation for each length-ℓ fragment of T, resulting in an $\mathcal{O}(n\ell)$-time algorithm, which is reasonably fast for modest ℓ. (Booth's algorithm gives the leftmost minimal rotation by construction.) We instead give an optimal $\mathcal{O}(n)$-time algorithm for the construction of $\mathcal{A}_\ell$, which is mostly of theoretical interest.

For every string X and every natural number m, we define the mth *power* of the string X, denoted by X^m , by $X^0 = \varepsilon$ and $X^k = X^{k-1}X$ for $k = 1, 2, \ldots, m$. A nonempty string is *primitive*, if it is not the power of any other string. Let us state two well-known combinatorial lemmas.

Lemma 1 ([17])**.** *A nonempty string X is primitive if and only if it occurs as a substring in XX only as a prefix and as a suffix.*

Example 5 (Illustration of Lemma 1)**.** String $X = \texttt{abb}$ is primitive because $\texttt{abb}$ occurs only as a prefix and as a suffix in $XX = \underline{\texttt{abb}} \cdot \underline{\texttt{abb}}$. String $Y = \texttt{aa}$ is not primitive because $\texttt{aa}$ occurs not only as a prefix and as a suffix in $YY = \texttt{a}\underline{\texttt{aa}}\texttt{a}$.

Lemma 2 ([74])**.** *Let $X = UV$ and $R = VU$, for two strings U, V. If X is primitive, then R is also primitive.*

Example 6 (Illustration of Lemma 2)**.** Let $U = \texttt{ab}$ and $V = \texttt{bb}$. Then $X = UV = \texttt{abbb}$ is primitive because $\texttt{abbb}$ occurs only as a prefix and as a suffix in $XX = \underline{\texttt{abbb}} \cdot \underline{\texttt{abbb}}$. Then $R = VU = \texttt{bbab}$ is also primitive because $\texttt{bbab}$ occurs only as a prefix and as a suffix in $RR = \underline{\texttt{bbab}} \cdot \underline{\texttt{bbab}}$.

A substring U of a string X is called an *infix* of X if and only if $U = X[i\,..\,j]$ with $i > 1$ and $j < n$.

Lemma 3. *A string X has more than one minimal lexicographic rotation if and only if X is a power of some string.*

Proof. ($\Rightarrow$) Let $X = U_1V_1$, and $R = V_1U_1$ be the leftmost minimal lexicographic rotation of X. Suppose towards a contradiction that X has another minimal lexicographic rotation but X is primitive. In particular, there exists $H = V_2U_2 = R$, with $X = U_2V_2$ and $|U_1| < |U_2|$. If X is primitive, then R is also primitive by Lemma 2 but then $RR = V_1U_1V_1U_1$ has H occurring as infix. In particular, in RR, V_2 is a suffix of the first occurrence of V_1 and U_2 is a prefix of U_1V_1 and thus $H = R$ occurs as infix. By Lemma 1 we obtain a contradiction.

($\Leftarrow$) Let $X = UU \cdots U$ and a minimal lexicographic rotation of X be $i \in [1, |X|]$. Then either $i + |U|$ or $i - |U|$ is a minimal lexicographic rotation of X.

□

Example 7 (Illustration of Lemma 3)**.** Let $X = \texttt{cbacbacba}$, $R = \texttt{acbacba} \cdot \texttt{cb}$ with $U_1 = \texttt{acbacba}$ and $V_1 = \texttt{cb}$, and $H = \texttt{acba} \cdot \texttt{cbacb} = R$ with $U_2 = \texttt{acba}$ and $V_2 = \texttt{cbacb}$. Observe that, in $RR = \texttt{acb}\underline{\texttt{acbacbacbac}}\texttt{bacbacb}$, H occurs as infix (shown as underlined) hence X is a power of some string.

Lemma 4. *Let X be a string of length n and set $Y = XX\#$, for some letter $\#$ not occurring in X that is the lexicographically maximal letter occurring in Y. Further, let $Y[i\,..\,2n+1]$ be the lexicographically minimal suffix of Y, for some $i \in [1, 2n]$. The leftmost lexicographically minimal rotation of X is i.*

Proof. First note that $i \in [1, n]$ because $\#$ is the lexicographically maximal letter occurring in Y.

We consider two cases: (i) X is primitive; and (ii) X is power of some string. In the first case, X has one lexicographically minimal rotation by Lemma 3, and thus this is i. In the second case, X has more than one lexicographically minimal rotations, but because X is power of some string and $\#$ is the lexicographically maximal letter occurring in Y, i is the leftmost lexicographically minimal rotation of X. □

We employ the data structure of Kociumaka [52, Theorem 20] to obtain the following result.

Theorem 2. *The set $\mathcal{A}_\ell(T)$, for any $\ell > 0$ and any T of length n, can be constructed in $\mathcal{O}(n)$ time.*

Proof. The data structure of Kociumaka [52, Theorem 20] gives the minimal lexicographic suffix for any concatenation Y of k arbitrary fragments of a string S in $\mathcal{O}(k^2)$ time after an $\mathcal{O}(|S|)$ time preprocessing.

We set $S = T\#$, for some letter $\#$ that does not occur in T and is the lexicographically maximal letter occurring in S. For each fragment $T[i\,..\,i+\ell-1]$, we compute the minimal lexicographic suffix of string

$$\begin{aligned} Y =& S[i\,..\,i+\ell-1] \cdot S[i\,..\,i+\ell-1] \cdot S[n+1] \\ =& T[i\,..\,i+\ell-1] \cdot T[i\,..\,i+\ell-1] \cdot \#, \end{aligned}$$

where $k = 3$ in $\mathcal{O}(k^2) = \mathcal{O}(1)$ time. This suffix of Y is the minimal lexicographic rotation by Lemma 4. □

3.1.2 Space-Efficient Construction of $\mathcal{A}_\ell$

It should be clear that, in the best case, the size of $\mathcal{A}_\ell$ is in $\mathcal{O}(n/\ell)$ and this bound is tight. The construction of Theorem 2 requires $\mathcal{O}(n)$ space. Ideally, we would thus like to compute $\mathcal{A}_\ell$ efficiently using (strongly) sublinear space. We generalize Theorem 2 to the following result.

Theorem 3. *The set $\mathcal{A}_\ell(T)$, for any $\ell > 0$, any T of length n, and any constant $\epsilon \in (0, 1]$, can be constructed in $\mathcal{O}(n + n^{1-\epsilon}\ell)$ time using $\mathcal{O}(n^\epsilon + \ell + |\mathcal{A}_\ell|)$ space.*

Proof. We compute $\mathcal{A}_\ell(T[\lceil n^\epsilon(i-1)\rceil + 1\,..\,\max(\lceil n^\epsilon i\rceil + \ell, n)])$ for all $i \in [1, \lceil n^{1-\epsilon}\rceil]$ using the algorithm from Theorem 2 and output their union. For any constant $\epsilon \in (0, 1]$, the alphabet size $|\Sigma| = n^{\mathcal{O}(1)} = (n^\epsilon + \ell)^{\mathcal{O}(1)}$ is still polynomial in the length $n^\epsilon + \ell$ of the fragments, so computing one such anchor set takes $\mathcal{O}(n^\epsilon + \ell)$ time and space by Theorem 2. We delete each fragment (and the associated data structure) before processing the subsequent anchor set: it takes $\mathcal{O}(n + n^{1-\epsilon}\ell)$ time and $\mathcal{O}(n^\epsilon + \ell)$ additional space to construct $\mathcal{A}_\ell(T)$. □

3.1.3 Expected Size of $\mathcal{A}_\ell$

We next analyze the expected size of $\mathcal{A}_\ell(T)$. We first show that if $\log \ell$ grows no faster than $\log \sigma$, where σ is the size of the alphabet Σ, then the expected size of $\mathcal{A}_\ell$ is in $\mathcal{O}(n/\ell)$. Otherwise, we slightly amend the sampling process to ensure that the expected size of the sample is again in $\mathcal{O}(n/\ell)$.

Lemma 5. *If T 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 $\mathcal{A}_\ell(T)$ is in $\mathcal{O}(\frac{n \log \ell}{\ell \log \sigma} + \frac{n}{\ell})$.*

Proof. If $\ell = 1$, then $\mathcal{A}_\ell(T) = n$. If $\ell = 2$, then $\mathcal{A}_\ell(T) = 1 + (n-2)(2\sigma^2+1)/3\sigma^2$. Now suppose $\ell \geq 3$. We say that $T[i\,..\,i+\ell-1]$ introduces a new bd-anchor if there exists $j \in [1,\ell]$ such that j is the bd-anchor of $T[i\,..\,i+\ell-1]$, but $j+k$ is not the bd-anchor of $T[i-k\,..\,i-k+\ell-1]$ for all $k \in [1, \max(\ell-j, i-1)]$. Let $N_i(T)$ denote the event that $T[i\,..\,i+\ell-1]$ introduces a new bd-anchor. Since the letters are independent identically distributed, the probability $\mathbb{P}[N_i(T)]$ only depends on and is non-increasing in the number of preceding overlapping length-k substrings. Therefore, the following holds for the expected size of $\mathcal{A}_\ell(T)$:

$$\begin{aligned}\mathbb{E}[|\mathcal{A}_\ell(T)|] =& \mathbb{P}[N_1(T)] + \cdots + \mathbb{P}[N_{n-\ell+1}(T)] \\ \leq& 1 + (n-\ell)\mathbb{P}[N_2(T)].\end{aligned}$$

Let p be the length of the shortest prefix of the lexicographically minimal rotation of $T[2\,..\,\ell+1]$ which is strictly smaller than the same length prefix of any other rotation of $T[2\,..\,\ell+1]$.

Note that $\mathbb{P}[N_2(T)]$ is at most equal to the sum of the following probabilities:

$$\mathbb{P}[T[1\,..\,\ell] \text{ or } T[2\,..\,\ell+1] \text{ is a power of some string}] \quad (1)$$
$$\mathbb{P}[T[1\,..\,\ell] \text{ is primitive with bd-anchor } 1] \quad (2)$$
$$\mathbb{P}[T[2\,..\,\ell+1] \text{ is primitive with bd-anchor} > \ell - \frac{3\log \ell}{\log \sigma}] \quad (3)$$
$$\mathbb{P}[T[2\,..\,\ell+1] \text{ is primitive and } p \geq 3\log \ell / \log \sigma]. \quad (4)$$

To bound the probability in (1), note that

$$\begin{aligned}\mathbb{P}[\text{length-}\ell \text{ string is a power of some string}] &\leq \sum_{d<\ell, d|\ell} \sigma^{-(\ell-d)} \\ &\leq \sum_{d \leq \ell/2} \sigma^{-(\ell-d)} \\ &= \sigma^{1-\ell/2}/(\sigma-1).\end{aligned}$$

The probability in (2) is bounded by $1/\ell$ since each letter of a primitive length-ℓ string is equally likely to be the anchor. Similarly, the probability in (3) is bounded by $(3\log \ell/\log \sigma + 1)/\ell$. Finally, the probability in (4) is bounded by the probability that two prefixes of length $\lceil 3\log \ell/\log \sigma \rceil$ of rotations of $T[2\,..\,\ell+1]$ are equal, which is at most $\ell^2 \cdot \sigma^{-3\log \ell/\log \sigma} = 1/\ell$. It follows that

$$\begin{aligned}\mathbb{P}[N_2(T)] &\leq \frac{2\sigma^{1-\ell/2}}{(\sigma-1)} + 1/\ell + (3\log \ell/\log \sigma + 1)/\ell + 1/\ell \\ &= \mathcal{O}\left(\frac{\log \ell}{\ell \log \sigma} + \frac{1}{\ell}\right).\end{aligned}$$

We conclude that for any $\ell > 0$ the expected size of $\mathcal{A}_\ell(T)$ is in $\mathcal{O}(\frac{n\log \ell}{\ell \log \sigma} + \frac{n}{\ell})$. □

We define a reduced version of bd-anchors to ensure that the expected size of the sample is in $\mathcal{O}(n/\ell)$.

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

Lemma 6. *If T 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 $\mathcal{A}^{red}_\ell(T)$ with $r = \lceil 4\log \ell/\log \sigma \rceil$ is in $\mathcal{O}(n/\ell)$.*

Proof. If $\ell \in \{1, 2\}$, then $\mathcal{A}^{\text{red}}_\ell(T) \leq n$. Now suppose $\ell \geq 3$. Analogously to $N_i(T)$ in Lemma 5, we denote the event that $T[i\,..\,i+\ell-1]$ introduces a new reduced bd-anchor by $N^{\text{red}}_i(T)$. Again we find

$$\begin{aligned}\mathbb{E}\left[\left|\mathcal{A}^{\text{red}}_\ell(T)\right|\right] &= \mathbb{P}[N^{\text{red}}_1(T)] + \cdots + \mathbb{P}[N^{\text{red}}_{n-\ell+1}(T)] \\ &\leq 1 + (n-\ell)\mathbb{P}[N^{\text{red}}_2(T)].\end{aligned}$$

Let p^{red} be the length of the shortest prefix of the lexicographically minimal rotation $j_1 \in [1, \ell - r]$ of $T[2\,..\,\ell+1]$ which is strictly smaller than the same length prefix of any other rotation $j_2 \in [1, \ell - r] \setminus \{j_1\}$. Using a similar proof to that of Lemma 5 we find that $\mathbb{P}[N^{\text{red}}_2(T)]$ is at most equal to the sum of the following probabilities:

$$\mathbb{P}[T[1\,..\,\ell] \text{ or } T[2\,..\,\ell+1] \text{ is a power of some string}] \quad (5)$$
$$\mathbb{P}[T[1\,..\,\ell] \text{ is primitive with bd-anchor } 1] \quad (6)$$
$$\mathbb{P}[T[2\,..\,\ell+1] \text{ is primitive with bd-anchor } \ell - r + 1] \quad (7)$$
$$\mathbb{P}[T[2\,..\,\ell+1] \text{ is primitive and } p^{\text{red}}. \quad (8)$$

Thus, $\mathbb{P}[N^{\text{red}}_2(T)] \leq 2\sigma^{1-\ell/2}/(\sigma-1) + 1/(\ell-r) + 1/(\ell-r) + \ell^2/\sigma^r = 2/\ell + o(1/\ell)$. We conclude that for any $\ell > 0$ the expected size of $\mathcal{A}^{\text{red}}_\ell(T)$ is in $\mathcal{O}(n/\ell)$. □

In particular, if $\log \ell = \mathcal{O}(\log \sigma)$, we employ the sampling mechanism underlying $\mathcal{A}_\ell(T)$, otherwise (when $\log \ell = \Omega(\log \sigma)$) we employ the sampling mechanism underlying $\mathcal{A}^{\text{red}}_\ell(T)$ with $r = \lceil 4\log \ell/\log \sigma \rceil$. This ensures that the *expected size* of the residual sampling is always in $\mathcal{O}(n/\ell)$. We summarize our main result in the following theorem.

Theorem 4. *If T is a string of length n, randomly generated by a memoryless source with identical letter probabilities, then, for any integer $\ell > 0$, the expected size of $\mathcal{A}_\ell(T)$ or $\mathcal{A}^{red}_\ell(T)$ is $\mathcal{O}(n/\ell)$.*

Constructing $\mathcal{A}^{\text{red}}_\ell(T)$ in $\mathcal{O}(n)$ time requires a trivial modification in the construction underlying Theorem 2. For each fragment $T[i\,..\,i+\ell-1]$, instead of computing the minimal lexicographic suffix of string

$$\begin{aligned}Y &= S[i\,..\,i+\ell-1] \cdot S[i\,..\,i+\ell-1] \cdot S[n+1] \\ &= T[i\,..\,i+\ell-1] \cdot T[i\,..\,i+\ell-1] \cdot \#,\end{aligned}$$

we compute the minimal lexicographic suffix of string

$$\begin{aligned}Y^{\text{red}} &= S[i\,..\,i+\ell-1] \cdot S[i\,..\,i+\ell-1-r] \cdot S[n+1] \\ &= T[i\,..\,i+\ell-1] \cdot T[i\,..\,i+\ell-1-r] \cdot \#,\end{aligned}$$

in $\mathcal{O}(1)$ time. We also directly obtain the trade-off in Theorem 3 for constructing $\mathcal{A}_\ell^{\text{red}}(T)$.

3.1.4 Minimizing the Size of $\mathcal{A}_\ell$ is NP-hard

The number of bd-anchors depends on the lexicographic total order defined on Σ. We now prove that finding the total order which minimizes the number of bd-anchors is NP-hard using a reduction from minimum feedback arc set [47]. Let us start be defining this problem. Given a directed graph $G(V, E)$, a *feedback arc set* in G is a subset of E that contains at least one edge from every cycle in G. Removing these edges from G breaks all of the cycles, producing a directed acyclic graph. In the *minimum feedback arc set* problem, we are given $G(V, E)$, and we are asked to compute a smallest feedback arc set in G. The decision version of the problem takes an additional parameter k as input, and it asks whether all cycles can be broken by removing at most k edges from E. The decision version is NP-complete [47] and the optimization version is APX-hard [24].

Theorem 5. *Let T be a string of length n over alphabet Σ. Further, let $\ell > 0$ be an integer. Computing a total order $\leq$ on Σ which minimizes $|\mathcal{A}_\ell(T)|$ is NP-hard.*

Proof. Let $G = (\Sigma, A)$ be any instance of the minimum feedback arc set problem. We will construct a string $S \in \Sigma^*$ in polynomial time in the size of G such that finding a total order $\leq$ on Σ which minimizes $\mathcal{A}_4(S)$ corresponds to finding a minimum feedback arc set in G.

We start with an empty string S. For each edge $(a, b) \in A$, we append $(aabab)^{2|A|+1}$ to S. Observe that, if $a < b$, all the a's of $(aabab)^{2|A|+1}$ and none of its b's are order-4 bd-anchors, except possibly the first a and last b depending on the preceding and subsequent letters in S. Thus there are $6|A| + 2$ to $6|A| + 4$ order-4 bd-anchors in $(aabab)^{2|A|+1}$. If on the other hand $a > b$, we analogously find that there will be $4|A| + 1$ to $4|A| + 3$ order-4 bd-anchors in $(aabab)^{2|A|+1}$.

Let $d_\leq$ be the number of edges $(a, b) \in A$ such that $a < b$. The total number of order-4 bd-anchors in S is

$$|\mathcal{A}_4(S)| = |A| \cdot 2 \cdot (2|A| + 1) + d_\leq \cdot (2|A| + 1) + \epsilon,$$

with $\epsilon \in [-|A|, |A|]$. Therefore, minimizing the total number of order-4 bd-anchors in S is equivalent to finding an order $\leq$ on the set Σ of vertices of G which minimizes $d_\leq$.

Note that if we delete all edges $(a, b) \in A$ such that $a < b$, then the residual graph is acyclic. Moreover, for each acyclic graph there exists an order on the vertices such that $a > b$ for all $(a, b) \in A$. Therefore the minimal $d_\leq$ equals the size of the minimum feedback arc set.

We conclude that, since finding the size of the minimum feedback arc set is NP-hard, so is finding a total order $\leq$ on Σ which minimizes the total number of order-4 bd-anchors. □

3.2 Density Evaluation

We compare the density of bd-anchors and reduced bd-anchors, denoted by BDA and rBDA, respectively, to the density of minimizers, for different values of w and k such that $\ell = w + k - 1$. This is a fair comparison because $\ell = w + k - 1$ is the length of the fragments considered by both mechanisms. We implemented bd-anchors, the standard minimizers mechanism from [71], and the minimizers mechanism with robust winnowing from [73]. The standard minimizers and those with robust winnowing are referred to as STD and WIN, respectively. There exists a long line of research on improving the density of minimizers in special regimes (see Section 6 for details). We stress that most of these algorithms are designed, implemented, or optimized, *only for the DNA alphabet*. We have tested against two state-of-the-art tools employing such algorithms: Miniception [86] and PASHA [26]. The former did not give better results than STD or WIN for the tested (w, k) values; and the latter does not scale beyond $k = 16$ or with large alphabets. We have thus omitted these results. Table 2 summarizes our competitors.

TABLE 1: Datasets characteristics.

Dataset	Length n	Alphabet size $\|\Sigma\|$
DNA	200,000,000	4
XML	200,000,000	95
ENGLISH	200,000,000	224
PROTEINS	200,000,000	27
SOURCES	200,000,000	229

TABLE 2: Summary of competitors.

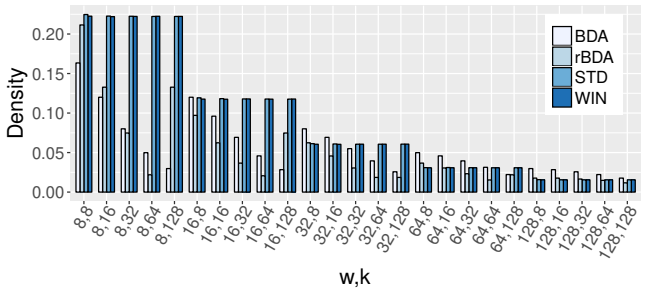
Competitor	Comment
STD [71]	Standard minimizers mechanism.
WIN [73]	Minimizers mechanism based on robust winnowing.
Miniception [86]	It works for DNA alphabet and is based on minimizers.
PASHA [26]	It works for DNA alphabet and is based on Universal Hitting Sets [68].

For bd-anchors, we used Booth's algorithm, which is easy to implement and reasonably fast. For minimizers, we used Karp-Rabin fingerprints [48]. (Note that such "random" minimizers tend to perform *even better* than the ones based on lexicographic total order in terms of density [86].) For the reduced version of bd-anchors, we used $r = \lceil 3 \log \ell / \log \sigma \rceil$, because the r value suggested by Lemma 6 is relatively large for the small ℓ values tested; e.g. for $\ell = w + k - 1 = 15$ and $\sigma = 4$, $\lceil 4 \log \ell / \log \sigma \rceil = 8$. Throughout, we do not evaluate construction times, as all implementations are reasonably fast, and we make the standard assumption that preprocessing is only required once. We used five string datasets from the popular Pizza & Chili corpus [29] (see Table 1 for the datasets characteristics).

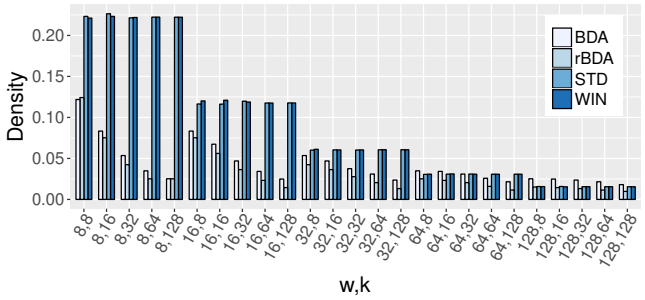
All implementations referred to in this paper have been written in `C++` and compiled at optimization level `-O3`. The source code of our implementations is available at https://github.com/solonas13/bd-anchors.

All experiments reported in this paper were conducted using a single core of an AMD Opteron 6386 SE 2.8GHz CPU and 252GB RAM running GNU/Linux.

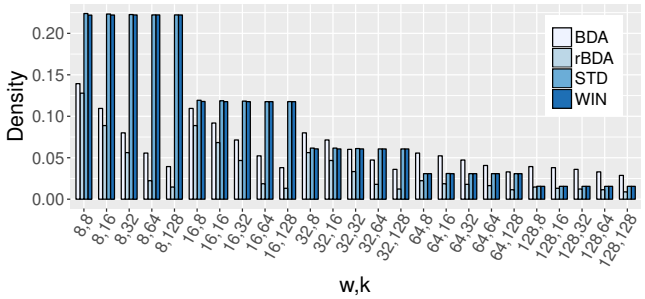
As can be seen by the results depicted in Figure 3, the density of both BDA and rBDA in the case of DNA, XML, and ENGLISH is either significantly smaller than or competitive to the STD and WIN minimizers density, especially for small w. Similar results for PROTEINS and SOURCES can be found in Supplemental Material. This is useful because a lower density results in smaller indexes and less computation (see Section 4), and because small w is of practical interest (see Section 5). For instance, the widely-used long-read aligner



(a) DNA



(b) XML



(c) ENGLISH

Fig. 3: Density vs. w, k for $\ell = w + k - 1$ and the first three datasets of Table 1.

Minimap2 [58] stores the selected minimizers of a reference genome in a hash table to find exact matches as anchors for seed-and-extend alignment. The parameters w and k are set based on the required sensitivity of the alignment, and thus w and k cannot be too large for high sensitivity. Thus, a lower sampling density reduces the size of the hash table, as well as the computation time, by lowering the average number of selected minimizers to consider when performing an alignment. Furthermore, although the datasets are not uniformly random, rBDA performs (often, significantly) better than BDA as ℓ grows, as suggested by Lemmas 5 and 6.

We next report the average number (AVG) of bd-anchors of order $\ell \in \{4, 8, 12, 16\}$ over all strings of length $n = 20$ (see Table 3a) and over all strings of length $n = 32$ (see Table 3b), both over a binary alphabet. Notably, the results show that AVG always lies in $[n/\ell, 2n/\ell]$ even if not using the reduced version of bd-anchors (see Lemma 6). As expected by Lemma 5, the analogous AVG values using a ternary alphabet (not reported here) were always lower than the corresponding ones with a binary alphabet.

TABLE 3: Average number of bd-anchors for varying ℓ and: (a) $n = 20$ and (b) $n = 32$.

(n, ℓ)	$(20, 4)$	$(20, 8)$	$(20, 12)$	$(20, 16)$
n/ℓ	5	2.5	1.66	1.25
AVG	8.53	4.37	2.77	1.76
$2n/\ell$	16	8	5.33	4

(a)

(n, ℓ)	$(32, 4)$	$(32, 8)$	$(32, 12)$	$(32, 16)$
n/ℓ	8	4	2,66	2
AVG	14.16	7.67	5.26	3.85
$2n/\ell$	16	8	5.33	4

(b)

4 Text Indexing

We show how bd-anchors can be applied to speed up on-line pattern searches. Let T be a string. We focus, in particular, on indexing T for answering the following type of on-line pattern search queries: Given a query string Q, return all occurrences of Q in T. In Section 4.1, we present our index and in Section 4.2 its experimental evaluation.

4.1 Index Construction and Querying

Before presenting our index, let us start with a basic definition that is central to our querying process.

Definition 3 ((α, β)-hit)**.** Given an order-ℓ bd-anchor $j_Q \in \mathcal{A}_\ell(Q)$, for some integer $\ell > 0$, of a query string Q, two integers $\alpha > 0, \beta > 0$, with $\alpha + \beta \geq \ell + 1$, and an order-$\ell$ bd-anchor $j_T \in \mathcal{A}_\ell(T)$ of a target string T, the ordered pair (j_Q, j_T) is called an (α, β)-*hit* if and only if $T[j_T - \alpha + 1 \,..\, j_T] = Q[j_Q - \alpha + 1 \,..\, j_Q]$ and $T[j_T \,..\, j_T + \beta - 1] = Q[j_Q \,..\, j_Q + \beta - 1]$.

Intuitively, the parameters α and β let us choose a fragment of Q that is anchored at j_Q.

Example 8. Let $T = \texttt{aabaaabcbda}$, $Q = \texttt{aacabaaaae}$, and $\ell = 5$. Consider that we would like to find the common fragment $Q[4 \,..\, 8] = T[2 \,..\, 6] = \texttt{abaaa}$. We know that the bd-anchor of order 5 corresponding to $Q[4 \,..\, 8]$ is $6 \in \mathcal{A}_5(Q)$, and thus to find it we set $\alpha = 3$ and $\beta = 3$. The ordered pair $(6, 4)$ is a $(3, 3)$-hit because for $4 \in \mathcal{A}_5(T)$, we have: $T[4{-}3{+}1 \,..\, 4] = Q[6{-}3{+}1 \,..\, 6] = \texttt{aba}$ and $T[4 \,..\, 4{+}3{-}1] = Q[6 \,..\, 6 + 3 - 1] = \texttt{aaa}$.

We would like to construct a data structure over T, which is based on $\mathcal{A}_\ell(T)$, such that, when we are given an order-ℓ bd-anchor j_Q over Q as an on-line query, together with parameters α and β, we can report all (α, β)-hits efficiently. To this end, we present an efficient data structure, denoted by $\mathcal{I}_\ell(T)$, which is constructed on top of T, and answers (α, β)-hit queries in near-optimal time. We prove the following result.

Theorem 6. *Given a string T of length n and an integer $\ell > 0$, the $\mathcal{I}_\ell(T)$ index can be constructed in $\mathcal{O}(n + |\mathcal{A}_\ell(T)|\sqrt{\log(|\mathcal{A}_\ell(T)|)})$ time. For any constant $\epsilon > 0$, $\mathcal{I}_\ell(T)$:*

- *occupies $\mathcal{O}(|\mathcal{A}_\ell(T)|)$ extra space and reports all k (α, β)-hits in $\mathcal{O}(\alpha + \beta + (k + 1)\log^\epsilon(|\mathcal{A}_\ell(T)|))$ time; or*
- *occupies $\mathcal{O}(|\mathcal{A}_\ell(T)|\log^\epsilon(|\mathcal{A}_\ell(T)|))$ extra space and reports all k (α, β)-hits in $\mathcal{O}(\alpha+\beta+\log\log(|\mathcal{A}_\ell(T)|)+k)$ time.*

Let us denote by $\overleftarrow{X} = X[|X|] \dots X[1]$ the *reversal* of string X. We now describe our data structure.

4.1.1 Construction of $\mathcal{I}_\ell(T)$

Given $\mathcal{A}_\ell(T)$, we construct two sets $\mathcal{S}^L_\ell(T)$ and $\mathcal{S}^R_\ell(T)$ of strings; conceptually, the reversed suffixes going *left* from j to 1, and the suffixes going *right* from j to n, for all j in $\mathcal{A}_\ell(T)$. In particular, for the bd-anchor j, we construct two strings: $\overleftarrow{T[1\,..\,j]} \in \mathcal{S}^L_\ell(T)$ and $T[j\,..\,n] \in \mathcal{S}^R_\ell(T)$. Note that, $|\mathcal{S}^L_\ell(T)| = |\mathcal{S}^R_\ell(T)| = |\mathcal{A}_\ell(T)|$, since for every bd-anchor in $\mathcal{A}_\ell(T)$ we have a distinct string in $\mathcal{S}^L_\ell(T)$ and in $\mathcal{S}^R_\ell(T)$.

We construct two *compacted tries* $\mathcal{T}^L_\ell(T)$ and $\mathcal{T}^R_\ell(T)$ over $\mathcal{S}^L_\ell(T)$ and $\mathcal{S}^R_\ell(T)$, respectively, to index all strings. Every string is concatenated with some special letter \$ not occurring in T, which is lexicographically minimal, to make $\mathcal{S}^L_\ell(T)$ and $\mathcal{S}^R_\ell(T)$ prefix-free (this is standard for conceptual convenience). The leaf nodes of the compacted tries are *labeled* with the corresponding j: there is a one-to-one correspondence between a leaf node and a bd-anchor j. In $\mathcal{O}(|\mathcal{A}_\ell(T)|)$ time, we also enhance the nodes of the tries with a perfect static dictionary [32] to ensure constant-time retrieval of edges by the first letter of their label. Let $\mathcal{L}^L_\ell(T)$ denote the list of the leaf labels of $\mathcal{T}^L_\ell(T)$ as they are visited using a depth-first search traversal. $\mathcal{L}^L_\ell(T)$ corresponds to the (labels of the) lexicographically sorted list of $\mathcal{S}^L_\ell(T)$ in increasing order. For each node u in $\mathcal{T}^L_\ell(T)$, we also store the corresponding interval $[x_u, y_u]$ over $\mathcal{L}^L_\ell(T)$. Analogously for R, $\mathcal{L}^R_\ell(T)$ denotes the list of the leaf labels of $\mathcal{T}^R_\ell(T)$ as they are visited using a depth-first search traversal and corresponds to the (labels of the) lexicographically sorted list of $\mathcal{S}^R_\ell(T)$ in increasing order. For each node v in $\mathcal{T}^R_\ell(T)$, we also store the corresponding interval $[x_v, y_v]$ over $\mathcal{L}^R_\ell(T)$.

The total size occupied by the tries is $\Theta(|\mathcal{A}_\ell(T)|)$ because they are compacted: we label the edges with intervals over $[1, n]$ from T.

We also construct a 2D range reporting data structure over the following points in set $\mathcal{R}_\ell(T)$:

$$(x, y) \in \mathcal{R}_\ell(T) \iff \mathcal{L}^L_\ell(T)[x] = \mathcal{L}^R_\ell(T)[y].$$

Note that $|\mathcal{R}_\ell(T)| = |\mathcal{A}_\ell(T)|$ because the set of leaf labels stored in both tries is precisely the set $\mathcal{A}_\ell(T)$. Let us remark that the idea of employing 2D range reporting for bidirectional pattern searches has been introduced by Amir et al. [3] for text indexing and dictionary matching with one error; see also [61].

This completes the construction of $\mathcal{I}_\ell(T)$. We next explain how we can query $\mathcal{I}_\ell(T)$.

4.1.2 Querying

Given a bd-anchor j_Q over a string Q as an on-line query and parameters $\alpha, \beta > 0$, we spell $\overleftarrow{Q[j_Q - \alpha + 1\,..\,j_Q]}$ in $\mathcal{T}^L_\ell(T)$ and $Q[j_Q\,..\,j_Q + \beta - 1]$ in $\mathcal{T}^R_\ell(T)$ starting from the root nodes. If any of the two strings is not spelled fully, we return no (α, β)-hits. If both strings are fully spelled, we arrive at node u in $\mathcal{T}^L_\ell(T)$ (resp. v in $\mathcal{T}^R_\ell(T)$), which corresponds to an interval over $\mathcal{L}^L_\ell(T)$ stored in u (resp. $\mathcal{L}^R_\ell(T)$ in v). We obtain the two intervals $[x_u, y_u]$ and $[x_v, y_v]$ forming a rectangle and ask the corresponding 2D range reporting query. It can be readily verified that this query returns all (α, β)-hits.

Example 9. Let T = `aabaaabcbda` and $\mathcal{A}_5(T) = \{4, 5, 6, 11\}$. We have the following strings in $\mathcal{S}^L(T)$: $\overleftarrow{T[1\,..\,4]}$ = `abaa`; $\overleftarrow{T[1\,..\,5]}$ = `aabaa`; $\overleftarrow{T[1\,..\,6]}$ = `aaabaa`; and $\overleftarrow{T[1\,..\,11]}$ = `adbcbaaabaa`. We have the following strings in $\mathcal{S}^R(T)$: $T[4\,..\,11]$ = `aaabcbda`; $T[5\,..\,11]$ = `aabcbda`; $T[6\,..\,11]$ = `abcbda`; $T[11\,..\,11]$ = `a`. Inspect Figure 4.

Proof of Theorem 6. We use the $\mathcal{O}(n)$-time algorithm underlying Theorem 2 to construct $\mathcal{A}_\ell(T)$. We use the $\mathcal{O}(n)$-time algorithm from [6], [12] to construct the compacted tries from $\mathcal{A}_\ell(T)$. We extract the $|\mathcal{A}_\ell(T)|$ points $(x, y) \in \mathcal{R}_\ell(T)$ using the compacted tries in $\mathcal{O}(|\mathcal{A}_\ell(T)|)$ time. For the first trade-off of the statement, we use the $\mathcal{O}(|\mathcal{A}_\ell(T)|\sqrt{\log(|\mathcal{A}_\ell(T)|)})$-time algorithm from [8] to construct the 2D range reporting data structure over $\mathcal{R}_\ell(T)$ from [11]. For the second trade-off, we use the $\mathcal{O}(|\mathcal{A}_\ell(T)|\sqrt{\log(|\mathcal{A}_\ell(T)|)})$-time algorithm from [34] to construct the 2D range reporting data structure over $\mathcal{R}_\ell(T)$ from the same paper. □

We obtain the following corollary for the fundamental problem of text indexing.

Corollary 7. *Given $\mathcal{I}_\ell(T)$ constructed for some integer $\ell > 0$ and some constant $\epsilon > 0$ over string T, we can report all k occurrences of any pattern Q, $|Q| \geq \ell$, in T in time:*

- *$\mathcal{O}(|Q| + (k+1)\log^\epsilon(|\mathcal{A}_\ell(T)|))$ when $\mathcal{I}_\ell(T)$ occupies $\mathcal{O}(|\mathcal{A}_\ell(T)|)$ extra space; or*
- *$\mathcal{O}(|Q| + \log\log(|\mathcal{A}_\ell(T)|) + k)$ when $\mathcal{I}_\ell(T)$ occupies $\mathcal{O}(|\mathcal{A}_\ell(T)|\log^\epsilon(|\mathcal{A}_\ell(T)|))$ extra space.*

Proof. Every occurrence of Q in T is prefixed by string $P = Q[1\,..\,\ell]$. We first compute the bd-anchor of P in $\mathcal{O}(\ell)$ time using Booth's algorithm. Let this bd-anchor be j. We set $\alpha = j$ and $\beta = |Q| - j + 1$. The result follows by applying Theorem 6. □

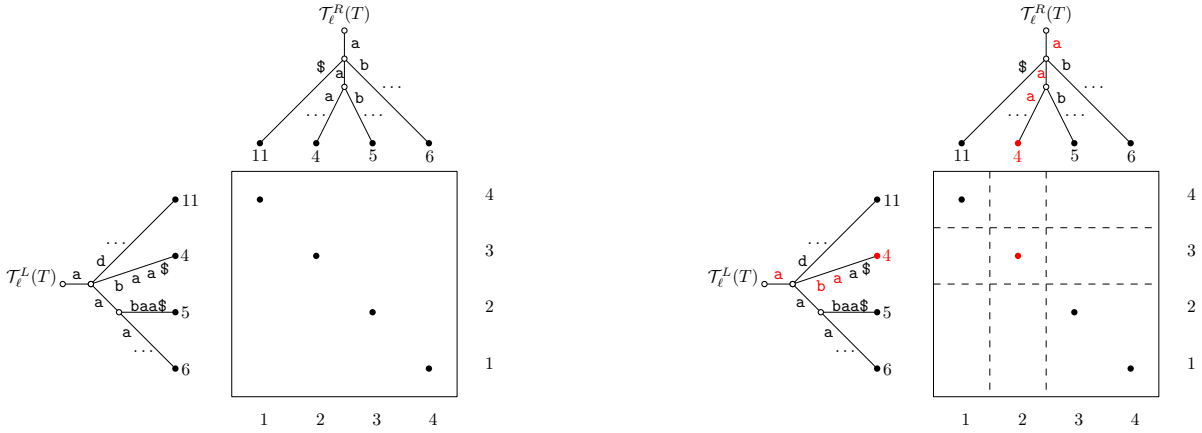
4.1.3 Querying Multiple Fragments

In the case of approximate pattern matching, we may want to query multiple length-ℓ fragments of a string Q given as an on-line query, and not only its length-ℓ prefix. We show that such an operation can be done efficiently using the bd-anchors of Q and the $\mathcal{I}_\ell(T)$ index.

Corollary 8. *Given $\mathcal{I}_\ell(T)$ constructed for some $\ell > 0$ and some constant $\epsilon > 0$ over T, for any sequence (not necessarily consecutive) of $d > 0$ length-ℓ fragments of a pattern Q, $|Q| \geq \ell$, corresponding to the same order-ℓ bd-anchor of Q, we can report all k_d occurrences of all d fragments in T in time:*

- *$\mathcal{O}(\ell + (d + k_d)\log^\epsilon(|\mathcal{A}_\ell(T)|))$ when $\mathcal{I}_\ell(T)$ occupies $\mathcal{O}(|\mathcal{A}_\ell(T)|)$ space; or*
- *$\mathcal{O}(\ell + d\log\log(|\mathcal{A}_\ell(T)|) + k_d)$ when $\mathcal{I}_\ell(T)$ occupies $\mathcal{O}(|\mathcal{A}_\ell(T)|\log^\epsilon(|\mathcal{A}_\ell(T)|))$ space.*

Proof. Let the order-ℓ bd-anchor over Q be j_Q and the corresponding parameters be $(\alpha_1, \beta_1), \cdots, (\alpha_d, \beta_d)$, with $\alpha_i + \beta_i = \ell + 1$. Observe that $\alpha_i > \alpha_{i+1}$ and $\beta_i < \beta_{i+1}$. Starting from j_Q, the string $\overleftarrow{Q[j_Q - \alpha_i + 1\,..\,j_Q]}$ we spell for fragment i is the prefix of $\overleftarrow{Q[j_Q - \alpha_{i-1} + 1\,..\,j_Q]}$ for fragment $i - 1$. The analogous property holds for the other direction: the string $Q[j_Q\,..\,j_Q + \beta_i]$ we spell for fragment i



(a) The $\mathcal{I}_\ell(T)$ index (b) Querying abaaa

Fig. 4: Let $T =$ aabaaabcbda and $\ell = 5$. Further let $Q =$ aacabaaaae, the bd-anchor $6 \in \mathcal{A}_5(Q)$ of order 5 corresponding to $Q[4\,..\,8]$, $\alpha = 3$ and $\beta = 3$. The figure illustrates the $\mathcal{I}_\ell(T)$ index and how we find that $Q[4\,..\,8] = T[2\,..\,5] =$ abaaa: the fragment $T[2\,..\,5]$ is anchored at position 4.

is the prefix of $Q[j_Q\,..\,j_Q + \beta_{i+1}]$ for fragment $i+1$. Thus it takes only $\mathcal{O}(\ell)$ time to construct all d rectangles. Finally, we ask the d corresponding 2D range reporting queries to obtain all k_d occurrences in the claimed time complexities. □

Example 10. Let $\ell = 5$, $T =$ aacaaabcbda, and $Q =$ aacaaaccbda. Strings T and Q have the same set of bd-anchors of order 5: $\mathcal{A}_5(T) = \mathcal{A}_5(Q) = \{4, 5, 6, 11\}$. Say we want to query $Q[1\,..\,5] =$ aacaa and $Q[3\,..\,7] =$ caaac *at the same time*, which both share 4 as their order-5 bd-anchor. The corresponding parameters are $(4, 2)$ (for $Q[1\,..\,5]$) and $(2, 4)$ (for $Q[3\,..\,7]$). Starting from position 4 of Q, the string $\overleftarrow{Q[4-2+1\,..\,4]} = \overleftarrow{Q[3\,..\,4]} =$ ac we spell for the second fragment is the prefix of $\overleftarrow{Q[4-4+1\,..\,4]} = \overleftarrow{Q[1\,..\,4]} =$ acaa for the first fragment. Thus we process both fragments at the same time. The analogous property holds for the other direction showing that $\mathcal{O}(\ell)$ time suffices for processing.

4.2 Index Evaluation

Consider a hash table with the following (key, value) pairs: the *key* is the hash value $h(S)$ of a length-k string S; and the *value* (satellite data) is a list of occurrences of S in T. It should be clear that such a hash table indexing the minimizers of T does not perform well for on-line pattern searches of *arbitrary length* because it would need to verify the remaining prefix and suffix of the pattern using letter comparisons *for all occurrences* of a minimizer in T. We thus opted for comparing our index to the one of [36], which addresses this specific problem by sampling the suffix array [62] with minimizers to reduce the number of letter comparisons during verification. We also compared against two classic text indexes: FM Index [28] and the suffix array (SA) [62]. Our implementations of these text indexes are based on SDSL [35], a mature library for succinct data structures. Table 4 summarizes the indexes we compared against.

To ensure a fair comparison, we have implemented the basic index from [36]; we denote it by GR Index. We used Karp-Rabin [48] fingerprints for computing the minimizers of T. We also used the array-based version of the suffix tree that consists of the suffix array (SA) and the longest common prefix (LCP) array [62]; SA was constructed using SDSL [35] and the LCP array using the Kasai et al. [49] algorithm.

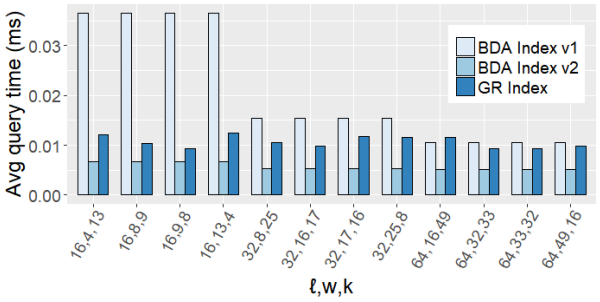
TABLE 4: Summary of competitors.

Competitor	Comment
GR Index [36]	Suffix array sampled with minimizers.
SA [62]	Suffix array.
FM Index [28]	Compressed text index based on the Burrows-Wheeler transform.

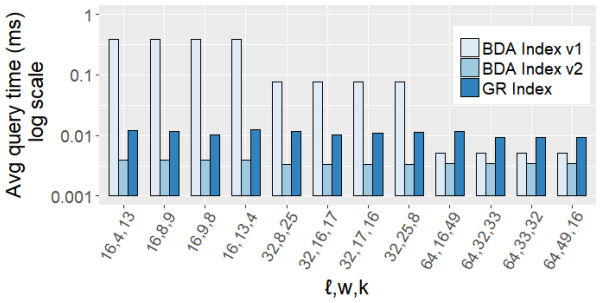
We sampled the SA using the minimizers. Given a pattern Q, we searched $Q[j\,..\,|Q|]$ starting with the minimizer $Q[j\,..\,j+k-1]$ using the Manber and Myers [62] algorithm on the sampled SA. For verifying the remaining prefix $Q[1\,..\,j-1]$ of Q, we used letter comparisons, as described in [36]. The space complexity of this implementation is $\mathcal{O}(n)$ and the extra space for the index is $\mathcal{O}(|\mathcal{M}_{w,k}(T)|)$. The query time is not bounded. We have implemented two versions of our index. We used Booth's algorithm for computing the bd-anchors of T. We used SDSL for SA construction and the Kasai et al. algorithm for LCP array construction. We sampled the SA using the bd-anchors thus constructing $\mathcal{L}_\ell^L(T)$ and $\mathcal{L}_\ell^R(T)$. Then, the two versions of our index are:

1) BDA Index v1: Let j be the bd-anchor of $Q[1\,..\,\ell]$. For $\overleftarrow{Q[1\,..\,j]}$ (resp. $Q[j\,..\,|Q|]$) we used the Manber and Myers algorithm for searching over $\mathcal{L}_\ell^L(T)$ (resp. $\mathcal{L}_\ell^R(T)$). We used range trees [9] implemented in CGAL [76] for 2D range reporting as per the described querying process. The space complexity of this implementation is $\mathcal{O}(n + |\mathcal{A}_\ell(T)|\log(|\mathcal{A}_\ell(T)|))$ and the extra space for the index is $\mathcal{O}(|\mathcal{A}_\ell(T)|\log(|\mathcal{A}_\ell(T)|))$. The query time is $\mathcal{O}(|Q| + \log^2(|\mathcal{A}_\ell(T)|) + k)$, where k is the total number of occurrences of Q in T.
2) BDA Index v2: Let j be the bd-anchor of $Q[1\,..\,\ell]$. If $|Q| - j + 1 \geq j$ (resp. $|Q| - j + 1 < j$), we search for $Q[j\,..\,|Q|]$ (resp. $\overleftarrow{Q[1\,..\,j]}$) using the Manber and Myers algorithm on $\mathcal{L}_\ell^R(T)$ (resp. $\mathcal{L}_\ell^L(T)$). For verifying the remaining part of the pattern we used letter comparisons. The space complexity of this implementation is $\mathcal{O}(n)$ and the extra space for the index is $\mathcal{O}(|\mathcal{A}_\ell(T)|)$. The

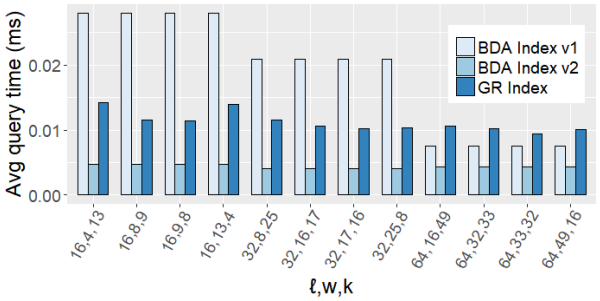
query time is not bounded.



(a) DNA

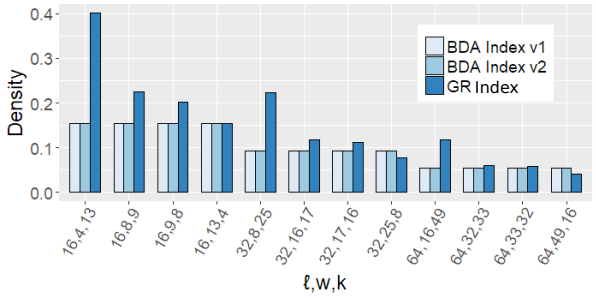


(b) XML

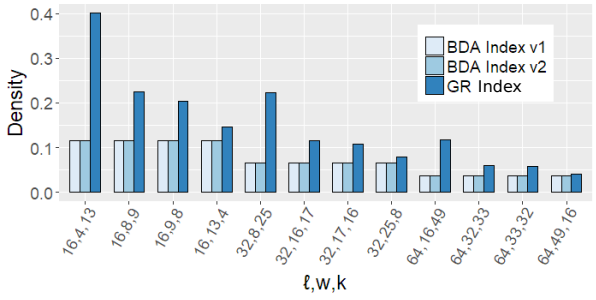


(c) ENGLISH

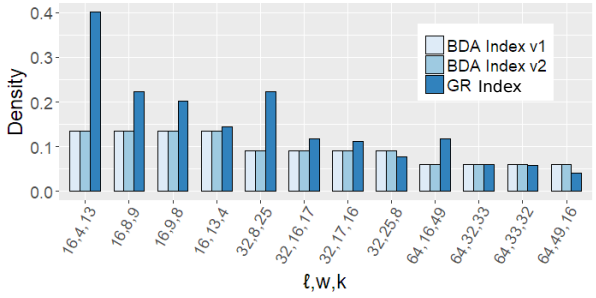
Fig. 5: Average query time (ms) vs. w, k for $\ell = w + k - 1$ and the first three datasets of Table 1.



(a) DNA



(b) XML



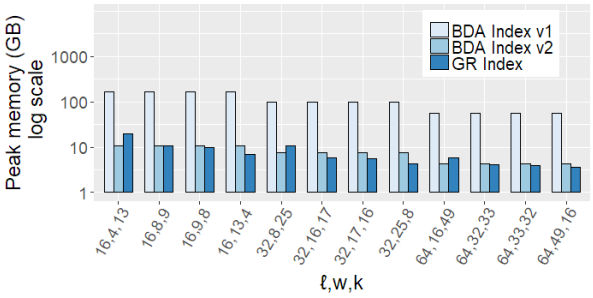
(c) ENGLISH

Fig. 6: Density vs. w, k for $\ell = w + k - 1$ and the first three datasets of Table 1.

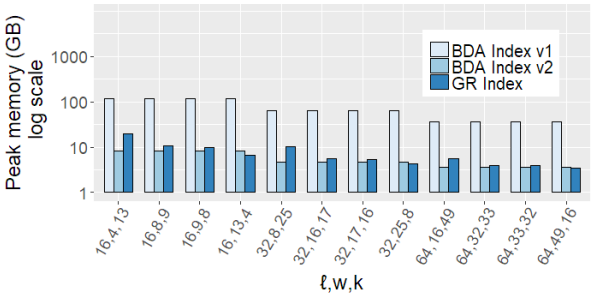
For each of the five real datasets of Table 1 and each query string length ℓ, we randomly extracted 500,000 substrings from the text and treated each substring as a query, following [36]. We plot the average query time in Figure 5 for the first three datasets of Table 1 (similar results for the other two datasets are in Supplemental Material). As can be seen, BDA Index v2 consistently outperforms GR Index across all datasets and all ℓ values. The better performance of BDA Index v2 is due to two theoretical reasons. First, the verification strategy exploits the fact that the index is *bidirectional* to apply the Manber and Myers algorithm to the largest part of the pattern, which results in fewer letter comparisons. Second, bd-anchors generally have smaller density compared to minimizers; see Figure 6 for the first three datasets of Table 1 and Supplemental Material for the rest. We also plot the peak memory usage; see Figure 7 for the first three datasets of Table 1 and Supplemental Material for the rest. As can be seen, BDA Index v2 requires a similar amount of memory to GR Index.

BDA Index v1 was slower than GR Index for small ℓ but faster for large ℓ in one of the two datasets used and had by far the highest memory usage. Let us stress that the inefficiency of BDA Index v1 is not due to inefficiency in the query time or space of our algorithm. It is merely because the range tree implementation of CGAL, which is a standard off-the-shelf library, is unfortunately inefficient in terms of both query time and memory usage; see also [75], [30].

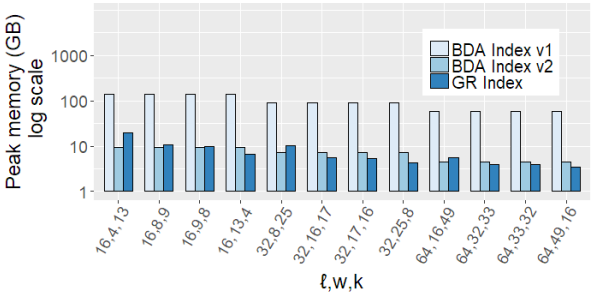
We now establish that BDA Index v2 offers much faster query time than the FM Index and the SA (see Figure 8). Specifically, BDA Index v2 outperformed the FM Index by up to 2 orders of magnitude, in the case of DNA that has a small alphabet (see Figure 8a), and by up to 3 to 4 orders of magnitude, in the case of PROTEINS and ENGLISH that have larger alphabets (see Figures 8b and 8c). BDA Index v2 was also faster than SA. This is remarkable since BDA Index v2 is a *sampled version* of SA; namely, we sample the suffix array using bd-anchors. Similar results for the other two datasets are in Supplemental Material. Our results are extremely encouraging, because both the FM Index and the SA are commonly used for text indexing and have also been incorporated into widely-used software [54], [59].



(a) DNA

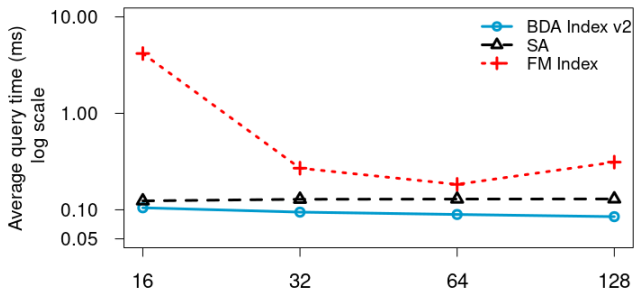


(b) XML

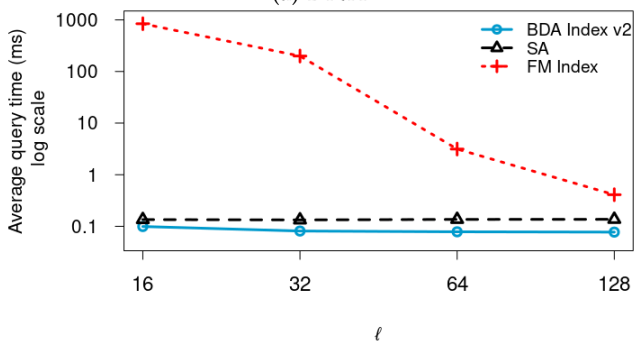


(c) ENGLISH

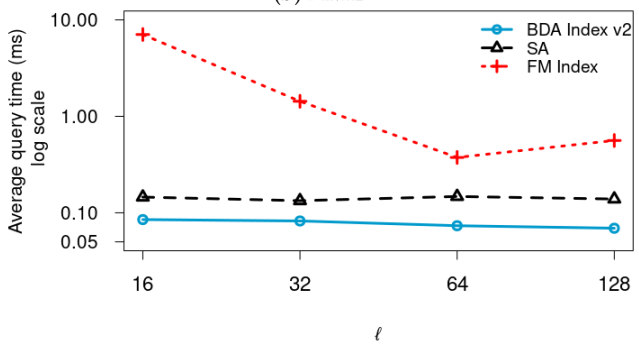
Fig. 7: Peak memory usage (GB) vs. w, k for $\ell = w + k - 1$ and the first three datasets of Table 1.



(a) DNA



(b) XML



(c) ENGLISH

Fig. 8: Average query time (ms) vs. ℓ for the first three datasets of Table 1.

4.2.1 Discussion

The proposed $\mathcal{I}_\ell(T)$ index, which is based on bd-anchors, has the following attributes:

1) **Construction**: $\mathcal{A}_\ell(T)$ is constructed in $\mathcal{O}(n)$ worst-case time and $\mathcal{I}_\ell(T)$ is constructed in $\mathcal{O}(n + |\mathcal{A}_\ell(T)|\sqrt{\log(|\mathcal{A}_\ell(T)|)})$ worst-case time. These time complexities are near-linear in n and do not depend on the alphabet Σ as long as $|\Sigma| = n^{\mathcal{O}(1)}$, which is true for virtually any real scenario.
2) **Index Size**: By Theorem 6, $\mathcal{I}_\ell(T)$ can occupy $\mathcal{O}(|\mathcal{A}_\ell(T)|)$ space. By Theorem 4, the size of $\mathcal{A}_\ell(T)$ (or $\mathcal{A}^{\text{red}}_\ell(T)$) is $\mathcal{O}(n/\ell)$ in expectation and so $\mathcal{I}_\ell(T)$ can also be of size $\mathcal{O}(n/\ell)$. In practice this depends on T and on the implementation of the 2D range reporting data structure.
3) **Querying**: The $\mathcal{I}_\ell(T)$ index answers on-line pattern searches in near-optimal time.
4) **Flexibility**: Note that one would have to *reconstruct* a (hash-based) index, which indexes the set of (w, k)-minimizers, to increase specificity or sensitivity: increasing k increases the specificity and decreases the sensitivity. Our $\mathcal{I}_\ell(T)$ index, conceptually truncated at string depth k, is essentially an index based on (w, k)-minimizers, which additionally wrap around. We can thus increase *specificity* by considering larger values for α and β or increase *sensitivity* by considering smaller values for α and β. This effect can be realized *without reconstructing* our $\mathcal{I}_\ell(T)$ index: we just adapt α and β upon querying accordingly.

5 Top-K Similarity Search under Edit Distance

We show how bd-anchors can be applied to speed up similarity search under edit distance. Let $\mathcal{D}$ be a collection of strings called *dictionary*. We focus, in particular, on indexing $\mathcal{D}$ for answering the following type of top-K queries: Given a query string Q and an integer K, return K strings from the dictionary that are closest to Q with respect to edit distance. In Section 5.1, we present our index and in Section 5.2 its experimental evaluation.

5.1 Index Construction and Querying

We follow a typical seed-chain-align approach as used by several bioinformatics applications [2], [19], [57], [58]. The main new ingredients we inject, with respect to this classic approach, is that we use: (1) bd-anchors as seeds; and (2) $\mathcal{I}_\ell$ to index the dictionary $\mathcal{D}$, for some integer parameter $\ell > 0$.

5.1.1 Construction

We require an integer parameter $\ell > 0$ defining the order of the bd-anchors. We set $T = S_1 \dots S_{|\mathcal{D}|}$, where $S_i \in \mathcal{D}$, compute the bd-anchors of order ℓ of T, and construct the $\mathcal{I}_\ell(T)$ index (see Section 4) using the bd-anchors.

5.1.2 Querying

We require two parameters $\tau \geq 0$ and $\delta \geq 0$. The former parameter controls the sensitivity of our filtering step (Step 2 below); and the latter one controls the sensitivity of our verification step (Step 3 below). Both parameters trade accuracy for speed.

1) For each query string Q, we compute the bd-anchors of order ℓ. For every bd-anchor j_Q, we take an arbitrary fragment (e.g. the leftmost) of length ℓ anchored at j_Q as the *seed*. Let this fragment start at position i_Q. This implies a value for α and β, with $\alpha+\beta = \ell+1$; specifically for $Q[i_Q \,..\, i_Q+\ell-1]$ we have $Q[i_Q \,..\, j_Q] = Q[j_Q-\alpha+1 \,..\, j_Q]$ and $Q[j_Q \,..\, i_Q+\ell-1] = Q[j_Q \,..\, j_Q+\beta-1]$. For every bd-anchor j_Q, we query $\overleftarrow{Q}[j_Q-\alpha+1 \,..\, j_Q]$ in $\mathcal{T}_\ell^L(T)$ and $Q[j_Q \,..\, j_Q+\beta-1]$ in $\mathcal{T}_\ell^R(T)$ and collect all (α,β)-hits.
2) Let $\tau \geq 0$ be an input parameter and let $L_{Q,S} = (q_1, s_1), \dots, (q_k, s_h)$ be the list of all (α,β)-hits between the queried fragments of string Q and fragments of a string $S \in \mathcal{D}$. If $h < \tau$, we consider string S as not found. The intuition here is that if Q and S are sufficiently close with respect to edit distance, they would have a relatively long $L_{Q,S}$ [19]. If $h \geq \tau$, we sort the elements of $L_{Q,S}$ with respect to their first component. (This comes for free because we process Q from left to right.) We then compute a *longest increasing subsequence* (LIS) in $L_{Q,S}$ with respect to the second component, which *chains* the (α,β)-hits, in $\mathcal{O}(h \log h)$ time [72] per $L_{Q,S}$ list. We use the LIS of $L_{Q,S}$ to *estimate* the *identity score* (total number of matching letters in a fixed alignment) for Q and S, which we denote by $E_{Q,S}$, based on the occurrences of the (α,β)-hits in the LIS.
3) Let $\delta \geq 0$ be an input parameter and let E_K be the Kth largest estimated identity score. We extract, as candidates, the ones whose estimated identity score is at least $E_K - \delta$. For every candidate string S, we close the gaps between the occurrences of the (α,β)-hits in the LIS using dynamic programming [55], thus computing an *upper bound* on the edit distance between Q and S (UB score). In particular, closing the gaps consists in summing up the exact edit distance for all pairs of fragments (one from S and one from Q) that lie in between the (α,β)-hits. We return K strings from the list of candidates with the lowest UB score. If $\delta = 0$, we return K strings with the highest $E_{Q,S}$ score.

5.2 Index Evaluation

We compared our algorithm, called BDA Search, to Min Search, the state-of-the-art tool for top-K similarity search under edit distance proposed by Zhang and Zhang in [84]. The main concept used in Min Search is the rank of a letter in a string, defined as the size of the neighborhood of the string in which the letter has the minimum hash value. Based on this concept, Min Search partitions each string in the dictionary $\mathcal{D}$ into a hierarchy of substrings and then builds an index comprised of a set of hash tables, so that strings having common substrings and thus small edit distance are grouped into the same hash table. To find the top-K closest strings to a query string, Min Search partitions the query string based on the ranks of its letters and then traverses the hash tables comprising the index. Thanks to the index and the use of several filtering tricks, Min Search is at least one order of magnitude faster with respect to query time than popular alternatives [83], [85], [21].

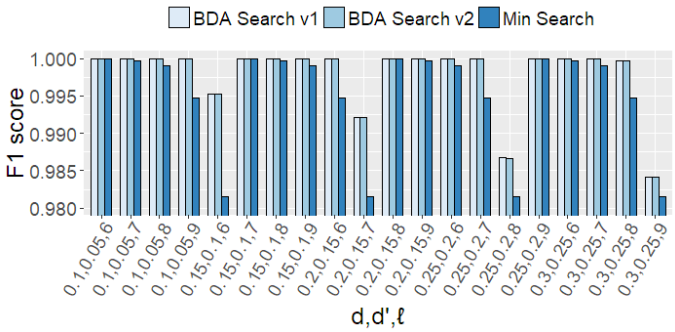
We implemented two versions of BDA Search: BDA Search v1 which is based on BDA Index v1; and BDA Search v2 which is based on BDA Index v2. For Min Search, we used the C++ implementation that is available in https://github.com/kedayuge/Search.

In the following, we present experiments on synthetic and on real data.

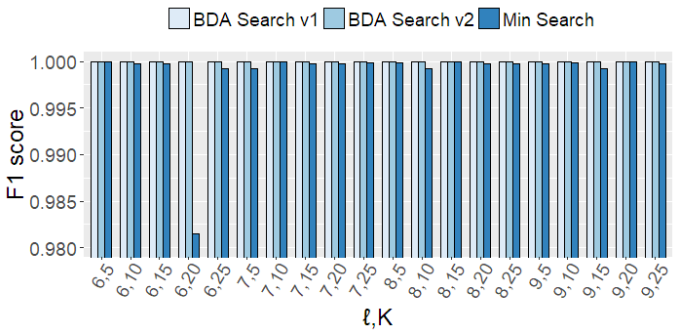
5.2.1 Synthetic Datasets

We constructed synthetic datasets, referred to as SYN, in a way that enables us to study the impact of different parameters and efficiently identify the ground truth (top-K closest strings to a query string with respect to edit distance). Specifically, we first generated 50 query strings and then constructed a cluster of K strings around each query string. To generate the query strings, we started from an arbitrary string Q of length $|Q| = 1000$ from a real dataset of protein sequences, used in [84], and generated a string Q' that is at edit distance e from Q, by performing e edit distance operations, each with equal probability. Then, we treated Q' as Q and repeated the process to generate the next query string. To create the clusters, we first added each query string into an initially empty cluster and then added $K-1$ strings, each at edit distance at most $e' < e$ from the query string. The strings were generated by performing at most e' edit distance operations, each with equal probability. Thus, each cluster contains the top-K closest strings to the query string of the cluster. We used $K \in \{5, 10, 15, 20, 25\}$, $d = \frac{e}{|Q|} \in \{0.1, 0.15, 0.2, 0.25, 0.3\}$, and $d' = \frac{e'}{|Q|} = d - 0.05$. We evaluated query answering accuracy using the F1 score [63], expressed as the harmonic mean of precision and recall[1]. For BDA Search, we report results for $\tau = 0$ (full sensitivity during filtering) and $\delta = 0$ (no sensitivity during verification), as it was empirically determined to be a reasonable trade-off between accuracy and speed. For Min Search, we report results using its default parameters from [84].

1. Precision is the ratio between the number of returned strings that are among the top-K closest strings to a query string and the number of all returned strings. Recall is the ratio between the number of returned strings that are among the top-K closest strings to a query string and K. Since all tested algorithms return K strings, the F1 score in our experiments is equal to precision and equal to recall.

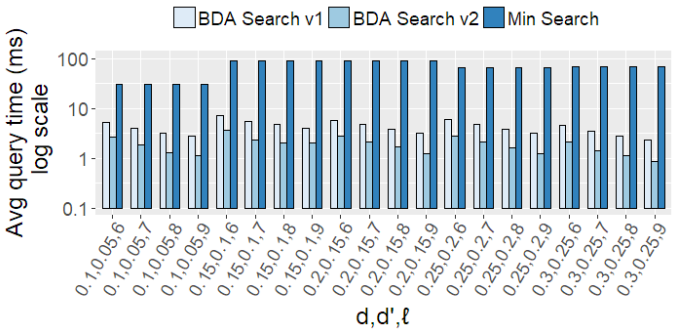


(a) SYN

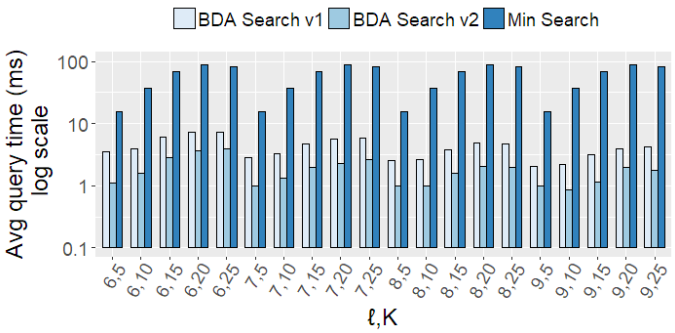


(b) SYN

Fig. 9: F1 score vs. (a) d, d', ℓ, for $K = 20$, and (b) ℓ, K, for $d = 0.15$ and $d' = 0.1$.



(a) SYN



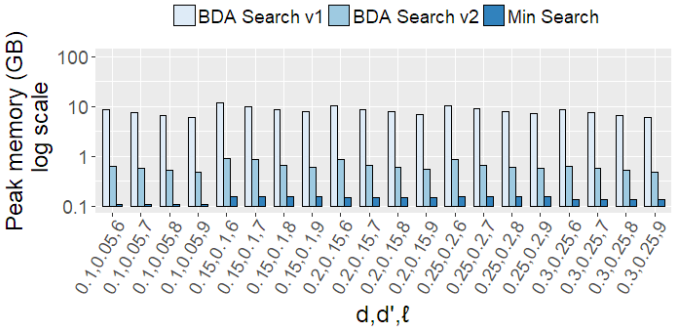
(b) SYN

Fig. 10: Average query time (ms) vs. (a) d, d', ℓ, for $K = 20$, and (b) ℓ, K, for $d = 0.15$ and $d' = 0.1$.

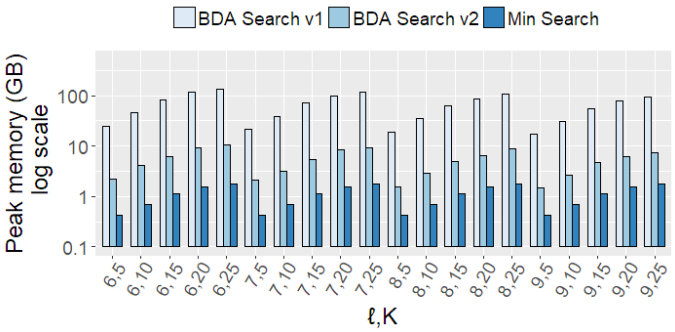
We plot the F1 scores and average query time in Figures 9 and 10, respectively. All methods achieved almost perfect accuracy, in all tested cases. BDA Search slightly outperformed Min Search (by up to 1.1%), remaining accurate even for large ℓ; the changes to F1 score for Min Search as ℓ varies are because the underlying method is randomized. However, both versions of BDA Search were *more than one order of magnitude faster* than Min Search on average (over all results of Figure 10), with BDA Search v1 being 2.9 times slower than BDA Search v2 on average, due to the inefficiency of the range tree implementation of CGAL. Furthermore, both versions of BDA Search scaled better with respect to K. For example, the average query time for BDA Search v1 became 2 times larger when K increased from 5 to 25 (on average over ℓ values), while that for Min Search became 5.4 times larger on average. The reason is that verification in Min Search, which increases the accuracy of this method, becomes increasingly expensive as K gets larger. The peak memory usage for these experiments is reported in Figure 11. Although Min Search outperforms BDA Search in terms of memory usage, BDA Search v2 still required a very small amount of memory (less than 1GB). BDA Search v1 required more memory for the reasons mentioned in Section 4.

5.2.2 Real Datasets

We used two real datasets containing genomic sequences. The first of these datasets, referred to as VIR, was obtained from [4] and contains: 116 Human rhinovirus genomes; 59 Ebola virus genomes; and 38 Influenza A virus genomes. The second dataset, referred to as GEN, was obtained from [84] and contains genomic sequences obtained by randomly



(a) SYN



(b) SYN

Fig. 11: Peak memory usage (GB) vs. (a) d, d', ℓ, for $K = 20$, and (b) ℓ, K, for $d = 0.15$ and $d' = 0.1$.

sampling substrings from Chromosome 20 of 50 Homo Sapiens individuals, and it was used in the evaluation of

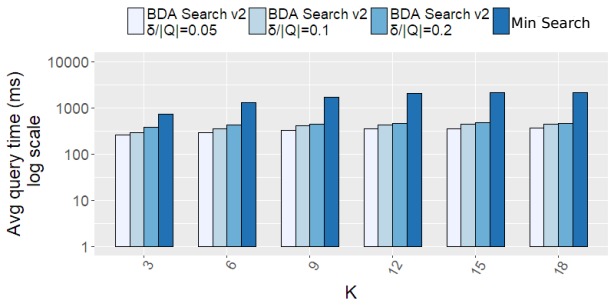
TABLE 5: Datasets characteristics.

Dataset	Dictionary size $\|\mathcal{D}\|$	Alphabet size $\|\Sigma\|$	Avg dictionary string length	Max dictionary string length
VIR	213	11	9,391	18,961
GEN	50,000	4	5,000	5,152

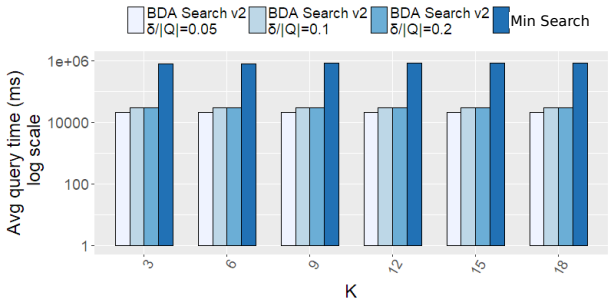
Min Search [84]. In the experiments with VIR, we used each string in this dataset as a query string. In the experiments with GEN, we used the first 10 queries from the query workload of [84] as query strings. Table 5 summarizes the characteristics of VIR and GEN. We compared BDA Search v2 to Min Search, since BDA Search v2 was equally effective but much faster than BDA Search v1. In our experiments, we specified a "normalized" δ per query string Q, dividing the input parameter δ of our method with the query length $|Q|$, to account for the different query lengths. We also fixed the order ℓ of bd-anchors to 12 and set parameter τ to 0; these values were empirically determined to offer an excellent trade-off between accuracy and efficiency, as it is demonstrated below.

We plot the average query time for varying K in Figure 12. BDA Search v2 outperformed Min Search for all K values and all datasets. The difference is in fact substantial. Specifically, our method was two to six times faster in VIR and more than one order of magnitude faster in GEN, which is a much larger dataset. These results are due to the use of bd-anchors; our method does not use any further filtering tricks unlike Min Search. Note also that increasing δ led to an increase in the average query time, since an edit distance upper bound is computed for more strings. However, the increase in the average query time was very small.

Figure 13 shows the total edit distance for all query answers in the experiment of Figure 12, for varying K. This measure simply sums up the edit distance between the query string and each of the K strings returned. It was used as a simple indicator of accuracy; it was computationally infeasible to compute the ground truth. As can be seen, our method was comparable to Min Search in terms of accuracy. In particular, the relative error between the total edit distance for all query answers for our method and that for Min Search was never larger than 0.5% in VIR and 0.9% in GEN (see Figure 14). Note that increasing δ helps accuracy but up to one point, since our method first finds a good set of candidates and then computes an upper bound of edit distance (Step 3) instead of the actual edit distance, which may introduce some error.

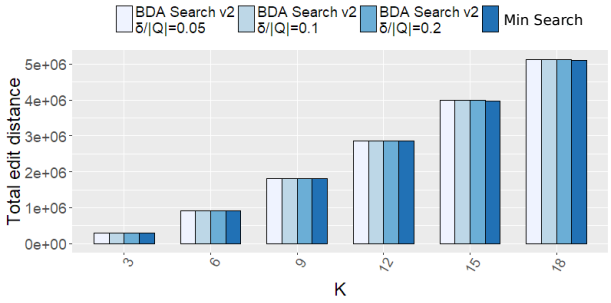


(a) VIR

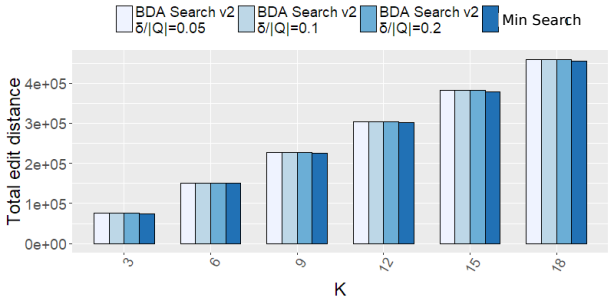


(b) GEN

Fig. 12: Average query time (ms) vs. K and δ for $\ell = 12$.



(a) VIR



(b) GEN

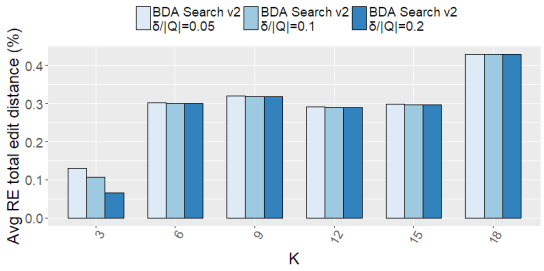
Fig. 13: Total edit distance vs. K and δ for $\ell = 12$.

5.2.3 Discussion

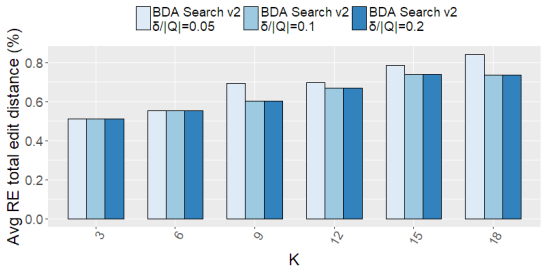
BDA Search offers comparable accuracy to that of Min Search but is faster by an order of magnitude or more in query time, especially for large dictionaries of long query strings. These results are very encouraging because the efficiency of BDA Search is entirely due to injecting bd-anchors and not due to any further filtering tricks such as those employed by Min Search. Min Search clearly outperforms BDA Search in memory usage, albeit the memory usage of BDA Search v2 is still quite modest.

6 Related Work

Although every sampling mechanism based on minimizers primarily aims at satisfying Properties 1 and 2, different mechanisms employ total orders that lead to substantially different total numbers of selected minimizers. Thus, research on minimizers has focused on determining total orders which lead to the lowest possible density (recall that the density is defined as the number of selected length-k substrings



(a) VIR



(b) GEN

Fig. 14: Average relative error of total edit distance (%) vs. K and δ for $\ell = 12$.

over the length of the input string). In fact, much of the literature focuses on the *average case* [68], [65], [64], [26], [86]; namely, the lowest expected density when the input string is random. In practice, many works use a "random minimizer" where the order is defined by choosing a permutation of all the length-k strings at random (e.g. by using a hash function, such as the Karp-Rabin fingerprints [48], on the length-k strings). Such a randomized mechanism has the benefit of being easy to implement and providing good expected performance in practice.

6.1 Minimizers and Universal Hitting Sets

A *universal hitting set* (UHS) is an unavoidable set of length-k strings, i.e., it is a set of length-k strings that "hits" every $(w + k - 1)$-long fragment of every possible string. The theory of universal hitting sets [68], [64], [50], [87] plays an important role in the current theory for minimizers with low density on average. In particular, if a UHS has small size, it generates minimizers with a provable upper-bound on their density. However, UHSs are less useful in the string-specific case for two reasons [89]: (1) the requirement that a UHS has to hit every $(w + k - 1)$-long fragment of every possible string is too strong; and (2) UHSs are too large to provide a meaningful upper-bound on the density in the string-specific case. Therefore, since in many practical scenarios the input string is known and does not change frequently, we try to optimize the density for one particular string instead of optimizing the average density over a random input.

In a recent work [88], Zheng et al. link UHSs to minimizers (as well as to other sampling schemes) with the motivation of designing sampling mechanisms with lower density and improving the algorithms that use such mechanisms.

6.2 String-Specific Minimizers

In the string-specific case, minimizers sampling mechanisms may employ frequency-based orders [14], [44]. In these orders, length-k strings occurring less frequently in the string compare less than the ones occurring more frequently. The intuition [89] is to obtain a sparse sampling by selecting infrequent length-k strings which should be spread apart in the string. However, there is no theoretical guarantee that a frequency-based order gives low density minimizers (there are many counter-examples). Furthermore, frequency-based orders do not always give minimizers with lower density in practice. For instance, the two-tier classification (very frequent vs. less frequent length-k strings) in the work of [44] outperforms an order that strictly follows frequency of occurrence.

A different approach to constructing string-specific minimizers is to start from a UHS and to remove elements from it, as long as it still hits every $(w + k - 1)$-long fragment of the input string [18]. Since this approach starts with a UHS that is not related to the string, the improvement in density may not be significant [89]. Additionally, current methods [26] employing this approach are computationally limited to using $k \leq 16$, as the size of the UHS increases exponentially with k. Using such small k values may not be appropriate in some applications.

In a recent work [39], Hoang et al. propose a deep-learning framework for learning string-specific minimizers.

6.3 Other Improvements

When $k \approx w$, minimizers with expected density of $1.67/w + o(1/w)$ on a random string can be constructed using the approach of [86]. Such minimizers have guaranteed expected density less than $2/(w + 1)$ and work for infinitely many w and k. The approach of [86] also does not require the use of expensive heuristics to precompute and store a large set of length-k strings, unlike some methods [68], [18], [26] with low density in practice.

The notion of *polar set*, which can be seen as complementary to that of UHS, was recently introduced in [89]. While a UHS is a set of length-k strings that intersect with every $(w + k - 1)$-long fragment at least once, a polar set is a set of length-k strings that intersect with any fragment at most once. The construction of a polar set builds upon sets of length-k strings that are sparse in the input string. Thus, the minimizers derived from these polar sets have provably tight bounds on their density. Unfortunately, computing optimal polar sets is NP-hard, as shown in [89]. Thus, the work of [89] also proposed a heuristic for computing feasible "good enough" polar sets. A main disadvantage of this approach is that when each length-k string occurs frequently in the input string, it becomes hard to select many length-k strings without violating the polar set condition.

Another notion related to minimizers is *syncmers*, introduced by Edgar in [25]. Syncmers are not string-specific and have been introduced as an alternative (more sensitive) sampling to minimizers, which is beneficial in DNA analysis.

7 Conclusion

We introduced bidirectional string anchors (bd-anchors, in short), a new string sampling mechanism. We showed that bd-anchors samples are approximately uniform, locally consistent, and computable in linear time. In addition, as we

demonstrated experimentally, the bd-anchors sample sizes decrease proportionally to ℓ; and these sizes are competitive to or smaller than the corresponding minimizers sample sizes. These results were theoretically justified by analyzing the expected size of bd-anchors samples. We also proved that computing a total order on the input alphabet which minimizes the bd-anchors sample size is NP-hard. Last, we highlighted the benefits of bd-anchors in two important applications: text indexing and top-K similarity search. For text indexing, we developed an index for performing on-line pattern searches in near-optimal time and showed experimentally that it is consistently faster for on-line pattern searches than commonly used indexes and an analogous implementation of a minimizers-based index. For top-K similarity search under edit distance, we developed a heuristic that is based on bd-anchors, and showed experimentally that it is generally as accurate as the state-of-the-art tool for the same purpose but more than one order of magnitude faster.

Acknowledgments

We would like to thank Tomasz Kociumaka for pointing us to [52, Theorem 20].

Michelle Sweering is supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003. This paper is part of the Leverhulme Trust RPG-2019-399 project.

This paper is also part of the PANGAIA project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement no. 872539. This paper is also part of the ALPACA project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement no. 956229.

Grigorios Loukides is an Associate Professor at King's College London. His research interests are in data privacy, data mining, and biomedical informatics.

Solon P. Pissis is a Senior Researcher at CWI and an Associate Professor at the Vrije Universiteit, both in Amsterdam. His research focuses on theory of algorithms and their application in data mining.

Michelle Sweering is a PhD student at CWI. Her research focuses on combinatorial algorithms on strings and graphs.

References

[1] Agrawal, R., Srikant, R.: Mining sequential patterns. In: ICDE, pp. 3–14 (1995)
[2] Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology **215**(3), 403–410 (1990)
[3] Amir, A., Keselman, D., Landau, G.M., Lewenstein, M., Lewenstein, N., Rodeh, M.: Text indexing and dictionary matching with one error. J. Algorithms **37**(2), 309–325 (2000)
[4] Anjum, N., Nabil, R., Rafi, R., Bayzid, S., Rahman, M.: CD-MAWS: An alignment-free phylogeny estimation method using cosine distance on minimal absent word sets. IEEE/ACM Transactions on Computational Biology and Bioinformatics (2021)
[5] Baeza-Yates, R., Ribeiro-Neto, B.A.: Modern Information Retrieval - the concepts and technology behind search, Second edition. Pearson Education Ltd (2011)
[6] Barton, C., Kociumaka, T., Liu, C., Pissis, S.P., Radoszewski, J.: Indexing weighted sequences: Neat and efficient. Inf. Comput. **270** (2020)
[7] Belazzougui, D.: Linear time construction of compressed text indices in compact space. In: STOC, pp. 148–193 (2014)
[8] Belazzougui, D., Puglisi, S.J.: Range predecessor and lempel-ziv parsing. In: SODA, pp. 2053–2071 (2016)
[9] de Berg, M., Cheong, O., van Kreveld, M.J., Overmars, M.H.: Computational geometry: algorithms and applications, 3rd Edition. Springer (2008)
[10] Booth, K.S.: Lexicographically least circular substrings. Inf. Process. Lett. **10**(4/5), 240–242 (1980)
[11] Chan, T.M., Larsen, K.G., Patrascu, M.: Orthogonal range searching on the RAM, revisited. In: SoCG, pp. 1–10 (2011)
[12] Charalampopoulos, P., Iliopoulos, C.S., Liu, C., Pissis, S.P.: Property suffix array with applications in indexing weighted sequences. ACM J. Exp. Algorithmics **25**, 1–16 (2020)
[13] Chaudhuri, S., Ganjam, K., Ganti, V., Motwani, R.: Robust and efficient fuzzy match for online data cleaning. In: SIGMOD, pp. 313–324 (2003)
[14] Chikhi, R., Limasset, A., Medvedev, P.: Compacting de Bruijn graphs from sequencing data quickly and in low memory. Bioinform. **32**(12), 201–208 (2016)
[15] Cole, R., Gottlieb, L., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: STOC, pp. 91–100. ACM (2004)
[16] Cole, R., Kopelowitz, T., Lewenstein, M.: Suffix trays and suffix trists: Structures for faster text indexing. Algorithmica **72**(2), 450–466 (2015)
[17] Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on strings. Cambridge University Press (2007)
[18] DeBlasio, D.F., Gbosibo, F., Kingsford, C., Marçais, G.: Practical universal k-mer sets for minimizer schemes. In: BCB, pp. 167–176 (2019)
[19] Delcher, A.L., Kasif, S., Fleischmann, R.D., Peterson, J., White, O., Salzberg, S.L.: Alignment of whole genomes. Nucleic Acids Research **27**(11), 2369–2376 (1999)
[20] Deng, D., Li, G., Feng, J.: A pivotal prefix based filtering algorithm for string similarity search. In: SIGMOD, pp. 673–684 (2014)
[21] Deng, D., Li, G., Feng, J., Li, W.: Top-k string similarity search with edit-distance constraints. In: ICDE, pp. 925–936 (2013)
[22] Deorowicz, S., Kokot, M., Grabowski, S., Debudaj-Grabysz, A.: KMC 2: fast and resource-frugal *k*-mer counting. Bioinform. **31**(10), 1569–1576 (2015)
[23] Dinklage, P., Fischer, J., Herlez, A., Kociumaka, T., Kurpicz, F.: Practical Performance of Space Efficient Data Structures for Longest Common Extensions. In: ESA 2020, pp. 39:1–39:20 (2020)
[24] Dinur, I., Safra, S.: The importance of being biased. In: STOC, pp. 33–42 (2002)
[25] Edgar, R.: Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences. PeerJ **9**, e10,805 (2021)
[26] Ekim, B., Berger, B., Orenstein, Y.: A randomized parallel algorithm for efficiently finding near-optimal universal hitting sets. In: RECOMB, pp. 37–53 (2020)
[27] Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS, pp. 137–143 (1997)
[28] Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005)
[29] Ferragina, P., Navarro, G.: Pizza&Chili corpus – compressed indexes and their testbeds. http://pizzachili.dcc.uchile.cl/texts.html

[30] Fisikopoulos, V.: An implementation of range trees with fractional cascading in C++. CoRR **abs/1103.4521** (2011)

[31] Flomin, D., Pellow, D., Shamir, R.: Data set-adaptive minimizer order reduces memory usage in *k*-mer counting. J. Comput. Biol. **29**(8), 825–838 (2022)

[32] Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with $\mathcal{O}(1)$ worst case access time. J. ACM **31**(3), 538–544 (1984)

[33] 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)

[34] Gao, Y., He, M., Nekrich, Y.: Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing. In: ESA, pp. 54:1–54:18 (2020)

[35] Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: SEA, pp. 326–337 (2014)

[36] Grabowski, S., Raniszewski, M.: Sampled suffix array with minimizers. Softw. Pract. Exp. **47**(11), 1755–1771 (2017)

[37] 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)

[38] Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press (1997)

[39] Hoang, M., Zheng, H., Kingsford, C.: Deepminimizer: A differentiable framework for optimizing sequence-specific minimizer schemes. In: RECOMB, pp. 52–69 (2022)

[40] Hon, W., Sadakane, K., Sung, W.: Breaking a time-and-space barrier in constructing full-text indices. SIAM J. Comput. **38**(6), 2162–2178 (2009)

[41] Hu, H., Li, G., Bao, Z., Feng, J., Wu, Y., Gong, Z., Xu, Y.: Top-k spatio-textual similarity join. IEEE TKDE **28**(2), 551–565 (2016)

[42] Jain, C., Dilthey, A.T., Koren, S., Aluru, S., Phillippy, A.M.: A fast approximate algorithm for mapping long reads to large reference databases. J. Comput. Biol. **25**(7), 766–779 (2018)

[43] Jain, C., Koren, S., Dilthey, A.T., Phillippy, A.M., Aluru, S.: A fast adaptive algorithm for computing whole-genome homology maps. Bioinform. **34**(17), i748–i756 (2018)

[44] Jain, C., Rhie, A., Zhang, H., Chu, C., Walenz, B., Koren, S., Phillippy, A.M.: Weighted minimizer sampling improves long read mapping. Bioinform. **36**(Supplement-1), i111–i118 (2020)

[45] Kahveci, T., Singh, A.K.: Efficient index structures for string databases. In: VLDB, pp. 351–360 (2001)

[46] Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006)

[47] Karp, R.M.: Reducibility among combinatorial problems. In: Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA, The IBM Research Symposia Series, pp. 85–103 (1972)

[48] Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987)

[49] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: CPM, pp. 181–192 (2001)

[50] Kempa, D., Kociumaka, T.: String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In: STOC, pp. 756–767 (2019)

[51] Koboldt, D.C., Steinberg, K.M., Larson, D.E., Wilson, R.K., Mardis, E.R.: The next-generation sequencing revolution and its impact on genomics. Cell **155**(1), 27–38 (2013)

[52] Kociumaka, T.: Minimal suffix and rotation of a substring in optimal time. In: CPM, pp. 28:1–28:12 (2016)

[53] Korf, I., Yandell, M., Bedell, J.A.: BLAST - an essential guide to the basic local alignment search tool. O'Reilly (2003)

[54] Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Biology **10**(3), R25 (2009)

[55] Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady **10**, 707 (1966)

[56] Li, C., Wang, B., Yang, X.: VGRAM: improving performance of approximate queries on string collections using variable-length grams. In: PVLDB, pp. 303–314 (2007)

[57] Li, H.: Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. Bioinformatics **32**(14), 2103–2110 (2016)

[58] Li, H.: Minimap2: pairwise alignment for nucleotide sequences. Bioinform. **34**(18), 3094–3100 (2018)

[59] Li, H., Durbin, R.: Fast and accurate short read alignment with burrows-wheeler transform. Bioinform. **25**(14), 1754–1760 (2009)

[60] Loukides, G., Pissis, S.P.: Bidirectional string anchors: A new string sampling mechanism. In: ESA, pp. 64:1–64:21 (2021)

[61] Mäkinen, V., Navarro, G.: Position-restricted substring searching. In: LATIN, pp. 703–714 (2006)

[62] Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)

[63] Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press (2008)

[64] Marçais, G., DeBlasio, D.F., Kingsford, C.: Asymptotically optimal minimizers schemes. Bioinform. **34**(13), i13–i22 (2018)

[65] Marçais, G., Pellow, D., Bork, D., Orenstein, Y., Shamir, R., Kingsford, C.: Improving the performance of minimizers and winnowing schemes. Bioinform. **33**(14), i110–i117 (2017)

[66] Munro, J.I., Navarro, G., Nekrich, Y.: Space-efficient construction of compressed indexes in deterministic linear time. In: SODA, pp. 408–424 (2017)

[67] Nyström-Persson, J., Keeble-Gagnère, G., Zawad, N.: Compact and evenly distributed k-mer binning for genomic sequences. Bioinform. **37**(17), 2563–2569 (2021)

[68] Orenstein, Y., Pellow, D., Marçais, G., Shamir, R., Kingsford, C.: Compact universal k-mer hitting sets. In: WABI, pp. 257–268 (2016)

[69] Qin, J., Wang, W., Xiao, C., Lu, Y., Lin, X., Wang, H.: Asymmetric signature schemes for efficient exact edit similarity query processing. ACM Trans. Database Syst. **38**(3), 16:1–16:44 (2013)

[70] Ramakrishnan, R., Gehrke, J.: Database management systems (3. ed.). McGraw-Hill (2003)

[71] 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)

[72] Schensted, C.: Longest increasing and decreasing subsequences. Canadian Journal of Mathematics **13**, 179–191 (1961)

[73] Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: Local algorithms for document fingerprinting. In: SIGMOD, pp. 76–85 (2003)

[74] Shyr, H., Thierrin, G.: Disjunctive languages and codes. In: FCT, pp. 171–176 (1977)

[75] Sun, Y., Blelloch, G.E.: Parallel range, segment and rectangle queries with augmented maps. In: ALENEX, pp. 159–173 (2019)

[76] The CGAL Project: CGAL User and Reference Manual, 5.2.1 edn. CGAL Editorial Board (2021). URL https://doc.cgal.org/5.2.1/Manual/packages.html

[77] Wang, J., Li, G., Feng, J.: Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In: SIGMOD, pp. 85–96 (2012)

[78] Wang, X., Ding, X., Tung, A.K.H., Zhang, Z.: Efficient and effective KNN sequence search with approximate n-grams. Proc. VLDB Endow. **7**(1), 1–12 (2013)

[79] Weiner, P.: Linear pattern matching algorithms. In: SWAT, pp. 1–11 (1973)

[80] Wood, D., Salzberg, S.: Kraken: Ultrafast metagenomic sequence classification using exact alignments. Genome Biology **15**(3) (2014)

[81] Yang, Z., Yu, J., Kitsuregawa, M.: Fast algorithms for top-k approximate string matching. In: AAAI, pp. 1467–1473 (2010)

[82] Ying, J.J., Lee, W., Weng, T., Tseng, V.S.: Semantic trajectory mining for location prediction. In: ACM SIGSPATIAL, pp. 34–43 (2011)

[83] Yu, M., Wang, J., Li, G., Zhang, Y., Deng, D., Feng, J.: A unified framework for string similarity search with edit-distance constraint. VLDB J. **26**(2), 249–274 (2017)

[84] Zhang, H., Zhang, Q.: Minsearch: An efficient algorithm for similarity search under edit distance. In: KDD, pp. 566–576 (2020)

[85] Zhang, Z., Hadjieleftheriou, M., Ooi, B.C., Srivastava, D.: Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In: SIGMOD, pp. 915–926 (2010)

[86] Zheng, H., Kingsford, C., Marçais, G.: Improved design and analysis of practical minimizers. Bioinform. **36**(Supplement-1), i119–i127 (2020)

[87] Zheng, H., Kingsford, C., Marçais, G.: Lower density selection schemes via small universal hitting sets with short remaining path length. In: RECOMB, pp. 202–217 (2020)

[88] Zheng, H., Kingsford, C., Marçais, G.: Lower density selection schemes via small universal hitting sets with short remaining path length. J. Comput. Biol. **28**(4), 395–409 (2021)

[89] Zheng, H., Kingsford, C., Marçais, G.: Sequence-specific minimizers via polar sets. Bioinform. **37**(Supplement), 187–195 (2021)