Bidirectional String Anchors: A New String Sampling Mechanism

Grigorios Loukides
Department of Informatics, King's College London, UK

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Abstract

The minimizers sampling mechanism is a popular mechanism for string sampling introduced independently by Schleimer et al. [SIGMOD 2003] and by Roberts et al. [*Bioinf.* 2004]. Given two positive integers w and k, it selects the lexicographically smallest length-k substring in every fragment of w consecutive length-k substrings (in every sliding window of length $w + k - 1$). Minimizers samples are approximately uniform, locally consistent, and computable in linear time. Although they do not have good worst-case guarantees on their size, they are often small in practice. They thus have been successfully employed in several string processing applications. Two main disadvantages of minimizers sampling mechanisms are: first, they also do not have good guarantees on the expected size of their samples for every combination of w and k; and, second, indexes that are constructed over their samples do not have good worst-case guarantees for on-line pattern searches.

To alleviate these disadvantages, we introduce bidirectional string anchors (bd-anchors), a new string sampling mechanism. Given a positive integer ℓ, our mechanism selects the lexicographically smallest rotation in every length-ℓ fragment (in every sliding window of length ℓ). We show that bd-anchors samples are also approximately uniform, locally consistent, and computable in linear time. In addition, our experiments using several datasets 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 using the analogous sampling parameters. We provide theoretical justification for these results by analyzing the expected size of bd-anchors samples.

We also show that by using any bd-anchors sample, we can construct, in near-linear time, an index which requires linear (extra) space in the size of the sample and answers on-line pattern searches in near-optimal time. We further show, using several datasets, that a simple implementation of our index is consistently faster for on-line pattern searches than an analogous implementation of a minimizers-based index [Grabowski and Raniszewski, *Softw. Pract. Exp.* 2017].

Finally, we highlight the applicability of bd-anchors by developing an efficient and effective heuristic for top-K similarity search under edit distance. We show, using synthetic datasets, that our heuristic is more accurate and more than one order of magnitude faster in top-K similarity searches than the state-of-the-art tool for the same purpose [Zhang and Zhang, KDD 2020].

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, string sampling, text indexing, top-K similarity search

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.64

Supplementary Material *Software (Source Code)*: `https://github.com/solonas13/bd-anchors`

Funding *Grigorios Loukides*: This paper is part of the Leverhulme Trust RPG-2019-399 project.
Solon P. Pissis: This paper is 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.

Acknowledgements We would like to thank Tomasz Kociumaka for pointing us to [42, Theorem 20] and Michelle Sweering for useful discussions.

© Grigorios Loukides and Solon P. Pissis;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

The notion of *minimizers*, introduced independently by Schleimer et al. [57] and by Roberts et al. [55], is a mechanism to sample a set of positions over an input string. 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.

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.

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 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 [45, 63, 19, 10, 30, 34, 35, 46, 36]. Since its inception, the minimizers sampling mechanism has undergone numerous theoretical and practical improvements [53, 10, 51, 50, 15, 21, 68, 36, 70] with a particular focus on minimizing the size of the residual sample; see Section 6 for a summary on this line of research. 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. $k = 16$ [21]. 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 to 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 $\mathcal{O}(n/w)$ [68]; 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, introduced by Kempa and Kociumaka [41], is another string sampling mechanism which may be employed to resolve the disadvantages of minimizers. Yet, it appears to be quite complicated to be efficient in practice. For instance, in [20], 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.

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 make the following three specific contributions:

1. In Section 3 we state that the set $\mathcal{A}_\ell(T)$, for any $\ell > 0$ and any T of length n, can be constructed in $\mathcal{O}(n)$ time; and 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 $\ell > 0$ (proofs are deferred to the full version of our work). 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 [68]. We then 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. In Section 4 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 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 [30].

3. In 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 a fundamental and extensively studied problem [37, 9, 11, 44, 64, 67, 54, 61, 60, 17, 33, 65, 66] with applications in areas including bioinformatics, databases, data mining, and information retrieval. We show, using synthetic datasets, that our heuristic, which is based on the bd-anchors index, is *more accurate* and *more than one order of magnitude faster* in top-K similarity searches than the state-of-the-art tool proposed by Zhang and Zhang in [66].

In Section 2, we provide some preliminaries; and in Section 6 we discuss works related to minimizers. Let us stress that, although other works may be related to our contributions, we focus on comparing to minimizers because they are extensively used in applications.

2 Preliminaries

We start with some basic definitions and notation following [13]. 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 $X[i \dots j]$ 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 $X[1 \dots j]$ and a *suffix* of X is a fragment of X of the form $X[i \dots n]$. 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 tree of a string of length n over an alphabet $\Sigma = \{1, \dots, n^{\mathcal{O}(1)}\}$ can be constructed in $\mathcal{O}(n)$ time [22].

Let us fix throughout a string $T = T[1 \dots n]$ 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 [55] (the definition in [57] is slightly different). Given an integer $k > 0$, an integer $w > 0$, and the ith length-$(w + k - 1)$ fragment $F = T[i \dots i + w + k - 2]$ 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 $T[i \dots i + w + k - 2]$, 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 [36]. This can be

implemented, for instance, using a rolling hash function (e.g. Karp-Rabin fingerprints [39]), 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 3.** *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 [22]. 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. [36]) over R. The set of reported indices is $\mathcal{M}_{w,k}(T)$. ◀

3 Bidirectional String Anchors

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 4** (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).

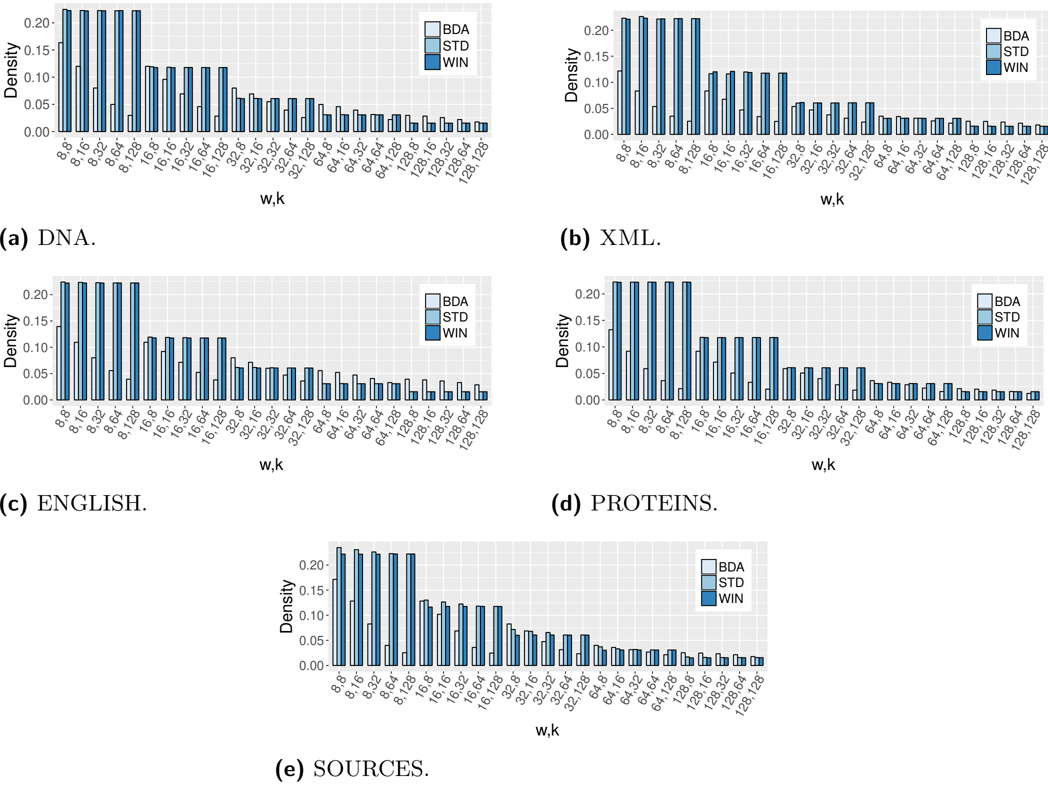
Construction and Size of $\mathcal{A}_\ell$. We show that $\mathcal{A}_\ell$ admits an efficient construction. One can use the linear-time algorithm by Booth [6] to compute the lexicographically minimal rotation for each length-ℓ fragment of T, resulting in an $\mathcal{O}(n\ell)$-time algorithm, which is reasonably fast for modest ℓ. (Booth's algorithm gives the leftmost minimal rotation by construction.) We can instead design an optimal $\mathcal{O}(n)$-time algorithm for the construction of $\mathcal{A}_\ell$, which is mostly of theoretical interest, via employing some elementary combinatorial observations and the data structure proposed by Kociumaka in [42, Theorem 20].

▶ **Theorem 5.** *The set $\mathcal{A}_\ell(T)$, for any integer $\ell > 0$ and any string T of length n, can be constructed in $\mathcal{O}(n)$ time.*

We can also show that the expected size of $\mathcal{A}_\ell(T)$ is in $\mathcal{O}(n/\ell)$ via observing that $\mathcal{A}_\ell(T)$ is expected to have many common elements with $\mathcal{M}_{w,k}(T)$, for certain values of w and k.

▶ **Theorem 6.** *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 $\mathcal{A}_\ell(T)$ is in $\mathcal{O}(n/\ell)$.*

We defer the proofs of Theorems 5 and 6 to the full version of our work.



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

Figure 1 Density vs. w, k for $\ell = w + k - 1$ and the datasets of Table 1.

Density Evaluation. We compare the density of bd-anchors, denoted by BDA, 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 [55], and the minimizers mechanism with robust winnowing from [57], which are referred to as STD and WIN, respectively.

For bd-anchors, we used Booth's algorithm, which is easy to implement and reasonably fast. For minimizers, we used Karp-Rabin fingerprints [39]. (Note that such "random" minimizers tend to perform *even better* than the ones based on lexicographic total order in terms of density [68].) 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 [24] (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`. 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 1, the density of bd-anchors is either significantly smaller than or competitive to the STD and WIN minimizers density, especially for small w. 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 Minimap2 [46] stores the selected minimizers of a reference genome in a hash table to find exact matches as anchors for seed-and-extend

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

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.

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 [68] and PASHA [21]. The former did not give better results than STD or WIN for the tested values of w and k; and the latter does not scale beyond $k = 16$ or with large alphabets. We have thus omitted these results.

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 2a) and over all strings of length $n = 32$ (see Table 2b), both over a binary alphabet. The results suggest that 2 may be a valid constant in $\mathcal{O}(n/\ell)$. As expected, the analogous AVG values using a ternary alphabet (not reported) were always lower than the corresponding ones with a binary alphabet.

Table 2 Average number of bd-anchors for varying ℓ and: (a) $n = 20$ and (b) $n = 32$.

(a)

(n, ℓ)	$(20, 4)$	$(20, 8)$	$(20, 12)$	$(20, 16)$
$2n/\ell$	10	5	3.33	2.5
AVG	8.53	4.37	2.77	1.76

(b)

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

4 Indexing Using Bidirectional Anchors

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

▶ **Definition 7** ((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 $T[j_T - \alpha + 1 \mathinner{.\,.} j_T] = Q[j_Q - \alpha + 1 \mathinner{.\,.} j_Q]$ and $T[j_T \mathinner{.\,.} j_T + \beta - 1] = Q[j_Q \mathinner{.\,.} j_Q + \beta - 1]$.*

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

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 8.** *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.

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: $\overleftarrow{T[1\mathinner{.\,.} j]} \in \mathcal{S}^L_\ell(T)$ and $T[j\mathinner{.\,.} n] \in \mathcal{S}^R_\ell(T)$. 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 [26] 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 $\mathcal{R}_\ell(T)$:

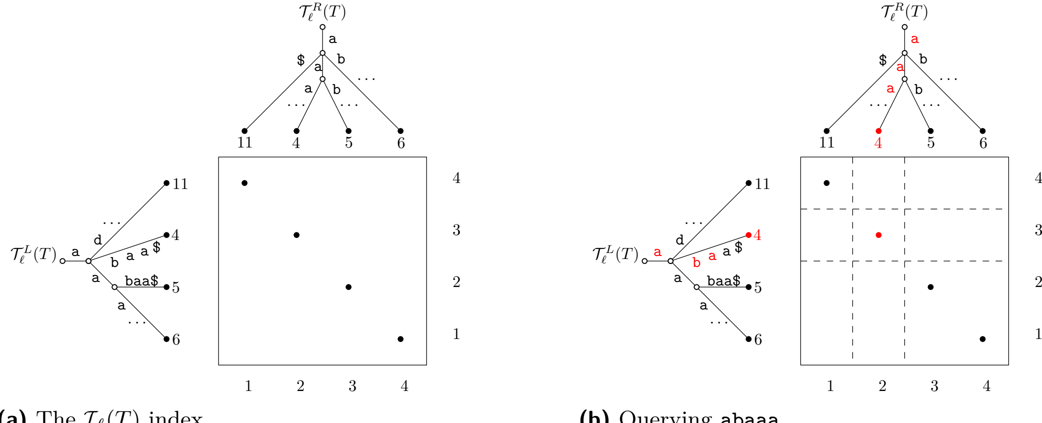
$$(x,y) \in \mathcal{R}_\ell(T) \iff \mathcal{L}^L_\ell(T)[x] = \mathcal{L}^R_\ell(T)[y].$$

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. [2] for text indexing and dictionary matching with one error; see also [47].

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

Querying. Given a bd-anchor j_Q over a string Q as an on-line query and parameters $\alpha, \beta > 0$, we spell $\overleftarrow{Q[j_Q-\alpha+1\mathinner{.\,.} j_Q]}$ in $\mathcal{T}^L_\ell(T)$ and $Q[j_Q\mathinner{.\,.} j_Q+\beta-1]$ 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 $T = \texttt{aabaaabcbda}$ and $\mathcal{A}_5(T) = \{4,5,6,11\}$. We have the following strings in $\mathcal{S}^L(T)$: $\overleftarrow{T[1\mathinner{.\,.} 4]} = \texttt{abaa}$; $\overleftarrow{T[1\mathinner{.\,.} 5]} = \texttt{aabaa}$; $\overleftarrow{T[1\mathinner{.\,.} 6]} = \texttt{aaabaa}$; and $\overleftarrow{T[1\mathinner{.\,.} 11]} = \texttt{adbcbaaabaa}$. We have the following strings in $\mathcal{S}^R(T)$: $T[4\mathinner{.\,.} 11] = \texttt{aaabcbda}$; $T[5\mathinner{.\,.} 11] = \texttt{aabcbda}$; $T[6\mathinner{.\,.} 11] = \texttt{abcbda}$; $T[11\mathinner{.\,.} 11] = \texttt{a}$. Inspect Figure 2.



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

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

Proof of Theorem 8. We use the $\mathcal{O}(n)$-time algorithm underlying Theorem 5 to construct $\mathcal{A}_\ell(T)$. We use the $\mathcal{O}(n)$-time algorithm from [3, 8] 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 [5] to construct the 2D range reporting data structure over $\mathcal{R}_\ell(T)$ from [7]. For the second trade-off, we use the $\mathcal{O}(|\mathcal{A}_\ell(T)|\sqrt{\log(|\mathcal{A}_\ell(T)|)})$-time algorithm from [28] 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* [62, 48, 22, 38, 23, 31, 32, 4, 12, 52, 41, 27].

▶ **Corollary 10.** *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 $P = Q[1\mathinner{.\,.}\ell]$. 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 8. ◀

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 [30], which addresses this specific problem by sampling the suffix array [48] with minimizers to reduce the number of letter comparisons during verification.

To ensure a fair comparison, we have implemented the basic index from [30]; we denote it by GR Index. We used Karp-Rabin [39] 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 [48]; SA was constructed using SDSL [29] and the LCP array using the Kasai et al. [40] algorithm.

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 [48] algorithm on the sampled SA. For verifying the remaining prefix $Q[1\,..\,j-1]$ of Q, we used letter comparisons, as described in [30]. 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 $\mathcal{L}^L_\ell(T)$ and $\mathcal{L}^R_\ell(T)$. Then, the two versions of our index are:

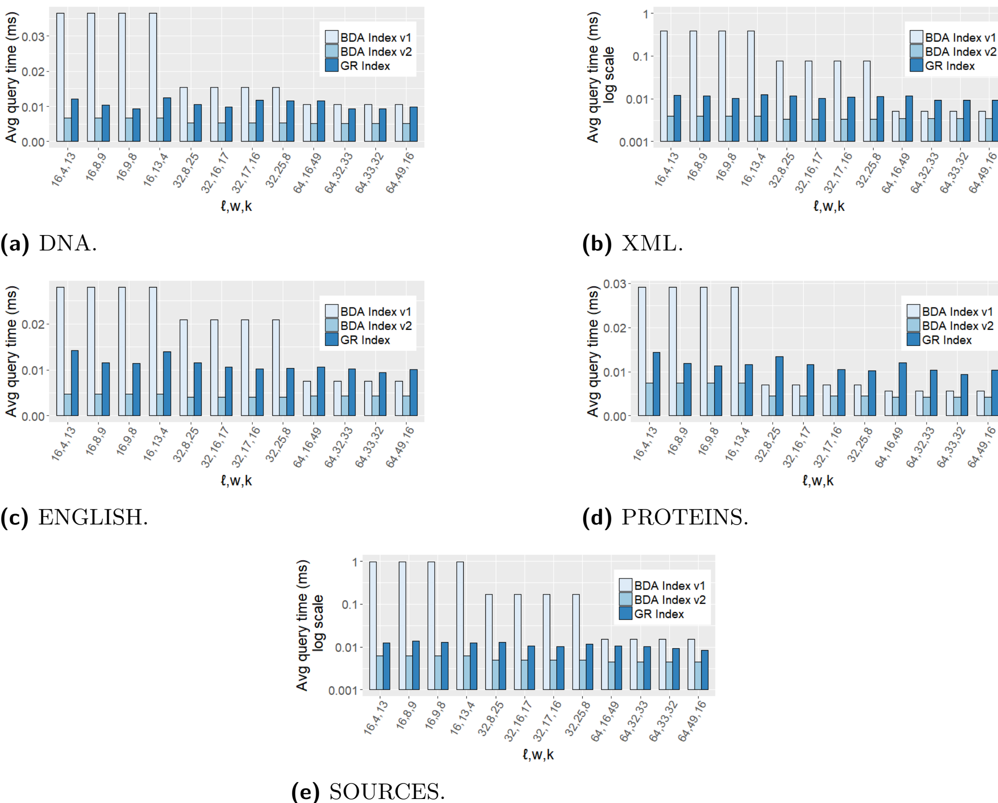
1. **BDA Index v1**: Let j be the bd-anchor of $Q[1\,..\,\ell]$. For $\overleftarrow{Q[1\,..\,j]}$ (resp. $Q[j\,..\,|Q|]$) we used the Manber and Myers algorithm for searching over $\mathcal{L}^L_\ell(T)$ (resp. $\mathcal{L}^R_\ell(T)$). We used range trees [14] implemented in CGAL [59] for 2D range 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 $\mathcal{L}^R_\ell(T)$ (resp. $\mathcal{L}^L_\ell(T)$). 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.

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 [30]. We plot the average query time in Figure 3. 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 4. We also plot the peak memory usage in Figure 5. 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 three out of five 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 [58, 25]. As BDA Index v2 was very efficient in all aspects, we defer the investigation of alternative range tree implementations [58] for BDA Index v1 to the full version of our work.

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 8, $\mathcal{I}_\ell(T)$ can occupy $\mathcal{O}(|\mathcal{A}_\ell(T)|)$ space. By Theorem 6, the size of $\mathcal{A}_\ell(T)$ 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.



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

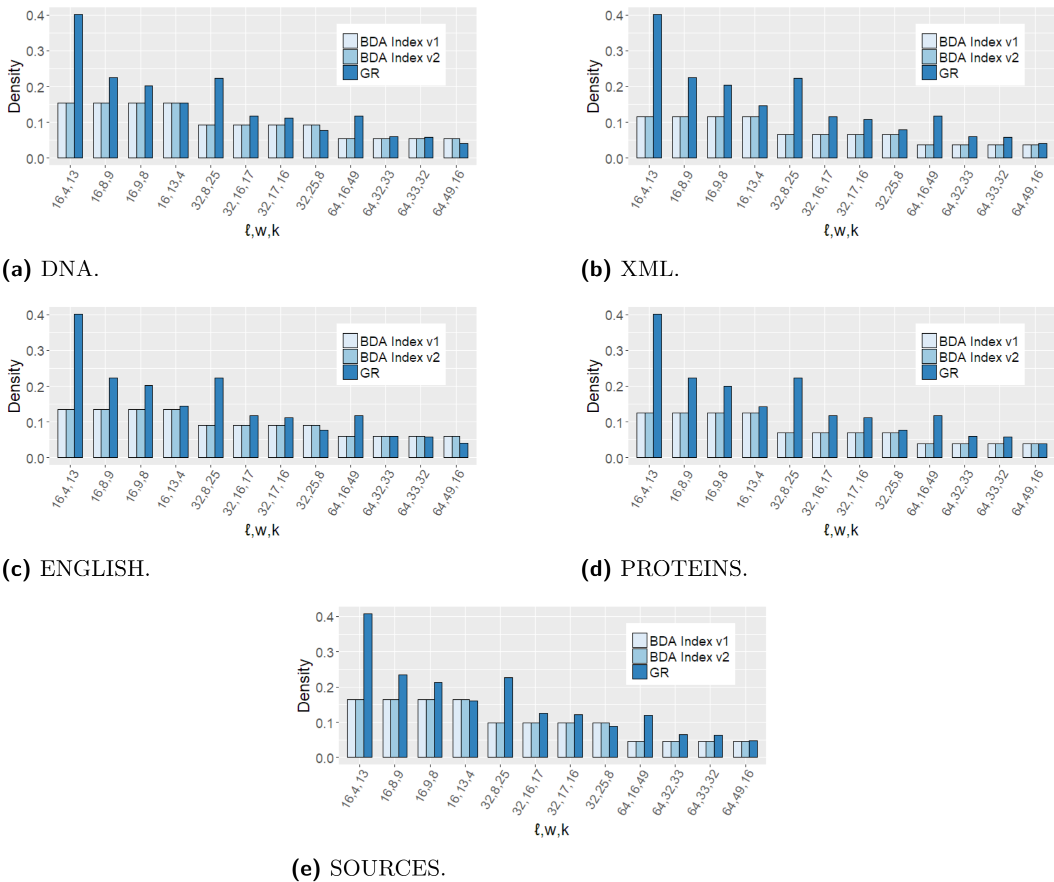
Figure 3 Average query time (ms) vs. w, k for $\ell = w + k - 1$ and the datasets of Table 1.

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 α, β values or increase *sensitivity* by considering smaller α, β values. 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. This is a fundamental problem with myriad applications in bioinformatics, databases, data mining, and information retrieval. It has thus been studied extensively in the literature both from a theoretical and a practical point of view [37, 9, 11, 44, 64, 67, 54, 61, 60, 17, 33, 65, 66]. 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. We follow a typical seed-chain-align approach as used by several bioinformatics applications [1, 16, 45, 46]. 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 $\ell > 0$.

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.

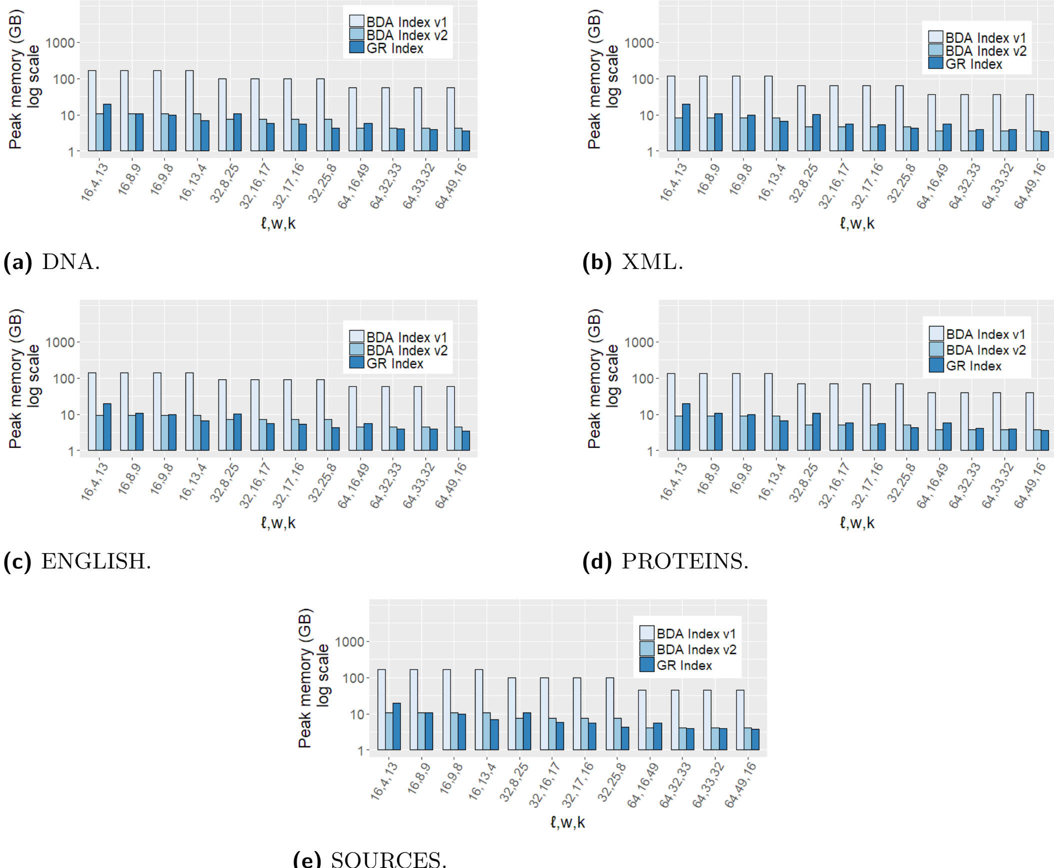


(a) DNA. **(b)** XML. **(c)** ENGLISH. **(d)** PROTEINS. **(e)** SOURCES.

Figure 4 Density vs. w, k for $\ell = w + k - 1$ and the datasets of Table 1.

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 $Q[i_Q \mathinner{.\,.} i_Q+\ell-1]$ we have $Q[i_Q \mathinner{.\,.} j_Q] = Q[j_Q-\alpha+1 \mathinner{.\,.} j_Q]$ and $Q[j_Q \mathinner{.\,.} i_Q + \ell - 1] = Q[j_Q \mathinner{.\,.} j_Q + \beta - 1]$. For every bd-anchor j_Q, we query $\overleftarrow{Q[j_Q - \alpha + 1 \mathinner{.\,.} j_Q]}$ in $\mathcal{T}^L_\ell(T)$ and $Q[j_Q \mathinner{.\,.} j_Q + \beta - 1]$ in $\mathcal{T}^R_\ell(T)$ 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 $L_{Q,S}$ [16]. 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 [56] 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.



(a) DNA.

(b) XML.

(c) ENGLISH.

(d) PROTEINS.

(e) SOURCES.

Figure 5 Peak memory usage (GB) vs. w, k for $\ell = w + k - 1$ and the datasets of Table 1.

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 [43], 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.

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 [66]. 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 [65, 67, 18].

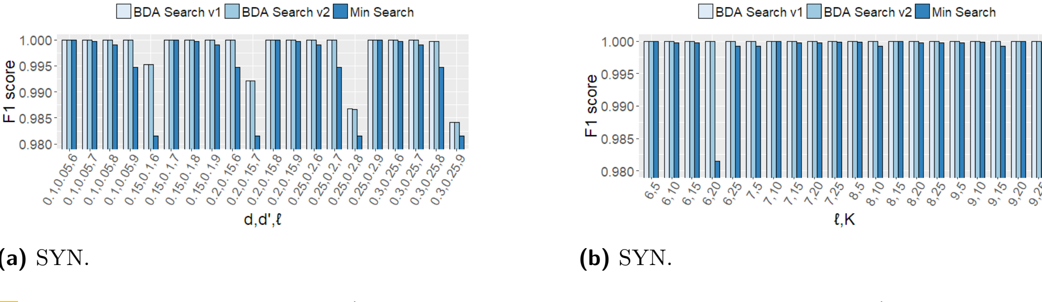
We implemented two versions of BDA Search: BDA Search v1 which is based on BDA Index v1; and BDA Search v2 which is based on BDA Index v2. For Min Search, we used the C++ implementation from `https://github.com/kedayuge/Search`.

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 [66], 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 [49], 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 [66].

We plot the F1 scores and average query time in Figures 6 and 7, 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 7), 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 8. 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.

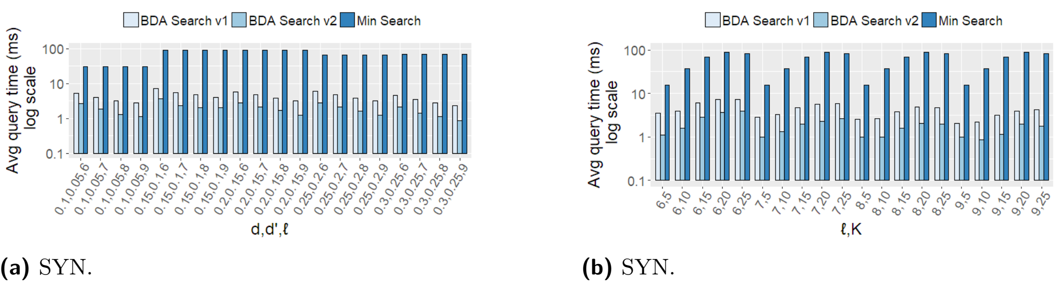
Discussion. BDA Search outperforms Min Search in accuracy while being more than one order of magnitude faster in query time. These results are very encouraging because the efficiency of BDA Search is entirely due to injecting bd-anchors and not due to any further filtering tricks such as those employed by Min Search. Min Search clearly outperforms BDA Search in memory usage, albeit the memory usage of BDA Search v2 is still quite modest. We defer an experimental evaluation using real datasets to the full version of our work.

[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.

Figure 6 F1 score vs. (a) d, d', ℓ, for $K = 20$, and (b) ℓ, K, for $d = 0.15$ and $d' = 0.1$.



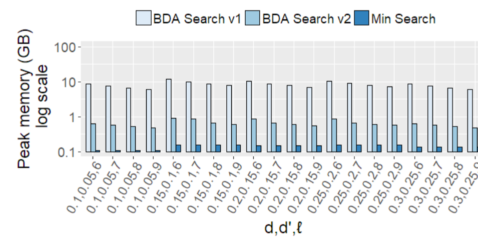
(a) SYN. **(b)** SYN.

Figure 7 Average query time (ms) vs. (a) d, d', ℓ, for $K = 20$, and (b) ℓ, K, for $d = 0.15$ and $d' = 0.1$.

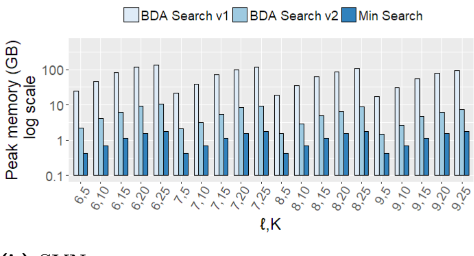
6 Other Works on Improving Minimizers

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 over the length of the input string). In fact, much of the literature focuses on the *average case* [53, 51, 50, 21, 68]; 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 [39], on the length-k strings). Such a randomized mechanism has the benefit of being easy to implement and providing good expected performance in practice.

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 [53, 50, 41, 69] 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 [70]: (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.



(a) SYN.



(b) SYN.

Figure 8 Peak memory usage (GB) vs. (a) d, d', ℓ, for $K = 20$, and (b) ℓ, K, for $d = 0.15$ and $d' = 0.1$.

String-Specific Minimizers. In the string-specific case, minimizers sampling mechanisms may employ frequency-based orders [10, 36]. In these orders, length-k strings occurring less frequently in the string compare less than the ones occurring more frequently. The intuition [70] 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 [36] 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 [15]. Since this approach starts with a UHS that is not related to the string, the improvement in density may not be significant [70]. Additionally, current methods [21] 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.

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 [68]. Such minimizers have guaranteed expected density less than $2/(w + 1)$ and work for infinitely many w and k. The approach of [68] also does not require the use of expensive heuristics to precompute and store a large set of length-k strings, unlike some methods [53, 15, 21] with low density in practice.

The notion of *polar set*, which can be seen as complementary to that of UHS, was recently introduced in [70]. 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 [70]. Thus, the work of [70] 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.

References

1 Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990. doi:10.1016/S0022-2836(05)80360-2.

2 Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, 2000. doi:10.1006/jagm.2000.1104.

3 Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P. Pissis, and Jakub Radoszewski. Indexing weighted sequences: Neat and efficient. *Inf. Comput.*, 270, 2020. doi:10.1016/j.ic.2019.104462.

4 Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 148–193, 2014. doi:10.1145/2591796.2591885.

5 Djamal Belazzougui and Simon J. Puglisi. Range predecessor and lempel-ziv parsing. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 2053–2071. SIAM, 2016. doi:10.1137/1.9781611974331.ch143.

6 Kellogg S. Booth. Lexicographically least circular substrings. *Inf. Process. Lett.*, 10(4/5):240–242, 1980. doi:10.1016/0020-0190(80)90149-0.

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

8 Panagiotis Charalampopoulos, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. Property suffix array with applications in indexing weighted sequences. *ACM J. Exp. Algorithmics*, 25, 2020. doi:10.1145/3385898.

9 Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 313–324. ACM, 2003. doi:10.1145/872757.872796.

10 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinform.*, 32(12):201–208, 2016. doi:10.1093/bioinformatics/btw279.

11 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 91–100. ACM, 2004. doi:10.1145/1007352.1007374.

12 Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix trays and suffix trists: Structures for faster text indexing. *Algorithmica*, 72(2):450–466, 2015. doi:10.1007/s00453-013-9860-6.

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

14 Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition.* Springer, 2008. URL: https://www.worldcat.org/oclc/227584184.

15 Dan F. DeBlasio, Fiyinfoluwa Gbosibo, Carl Kingsford, and Guillaume Marçais. Practical universal k-mer sets for minimizer schemes. In Xinghua Mindy Shi, Michael Buck, Jian Ma, and Pierangelo Veltri, editors, *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics, BCB 2019, Niagara Falls, NY, USA, September 7-10, 2019*, pages 167–176. ACM, 2019. doi:10.1145/3307339.3342144.

16 Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White, and Steven L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, January 1999. doi:10.1093/nar/27.11.2369.

17 Dong Deng, Guoliang Li, and Jianhua Feng. A pivotal prefix based filtering algorithm for string similarity search. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International*

Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014, pages 673–684. ACM, 2014. doi:10.1145/2588555.2593675.

18 Dong Deng, Guoliang Li, Jianhua Feng, and Wen-Syan Li. Top-k string similarity search with edit-distance constraints. In Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 925–936. IEEE Computer Society, 2013. doi:10.1109/ICDE.2013.6544886.

19 Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinform.*, 31(10):1569–1576, 2015. doi:10.1093/bioinformatics/btv022.

20 Patrick Dinklage, Johannes Fischer, Alexander Herlez, Tomasz Kociumaka, and Florian Kurpicz. Practical Performance of Space Efficient Data Structures for Longest Common Extensions. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:20, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2020.39.

21 Baris Ekim, Bonnie Berger, and Yaron Orenstein. A randomized parallel algorithm for efficiently finding near-optimal universal hitting sets. In Russell Schwartz, editor, *Research in Computational Molecular Biology - 24th Annual International Conference, RECOMB 2020, Padua, Italy, May 10-13, 2020, Proceedings*, volume 12074 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2020. doi:10.1007/978-3-030-45257-5_3.

22 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143, 1997. doi:10.1109/SFCS.1997.646102.

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

24 Paolo Ferragina and Gonzalo Navarro. Pizza&Chili corpus – compressed indexes and their testbeds. http://pizzachili.dcc.uchile.cl/texts.html.

25 Vissarion Fisikopoulos. An implementation of range trees with fractional cascading in C++. *CoRR*, abs/1103.4521, 2011. arXiv:1103.4521.

26 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $\mathcal{O}(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

27 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.

28 Younan Gao, Meng He, and Yakov Nekrich. Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 54:1–54:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2020.54.

29 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014. doi:10.1007/978-3-319-07959-2_28.

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

31 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

32 Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM J. Comput.*, 38(6):2162–2178, 2009. doi:10.1137/070685373.

33 Huiqi Hu, Guoliang Li, Zhifeng Bao, Jianhua Feng, Yongwei Wu, Zhiguo Gong, and Yaoqiang Xu. Top-k spatio-textual similarity join. *IEEE Trans. Knowl. Data Eng.*, 28(2):551–565, 2016. doi:10.1109/TKDE.2015.2485213.

34 Chirag Jain, Alexander T. Dilthey, Sergey Koren, Srinivas Aluru, and Adam M. Phillippy. A fast approximate algorithm for mapping long reads to large reference databases. *J. Comput. Biol.*, 25(7):766–779, 2018. doi:10.1089/cmb.2018.0036.

35 Chirag Jain, Sergey Koren, Alexander T. Dilthey, Adam M. Phillippy, and Srinivas Aluru. A fast adaptive algorithm for computing whole-genome homology maps. *Bioinform.*, 34(17):i748–i756, 2018. doi:10.1093/bioinformatics/bty597.

36 Chirag Jain, Arang Rhie, Haowen Zhang, Claudia Chu, Brian Walenz, Sergey Koren, and Adam M. Phillippy. Weighted minimizer sampling improves long read mapping. *Bioinform.*, 36(Supplement-1):i111–i118, 2020. doi:10.1093/bioinformatics/btaa435.

37 Tamer Kahveci and Ambuj K. Singh. Efficient index structures for string databases. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 351–360. Morgan Kaufmann, 2001. URL: http://www.vldb.org/conf/2001/P351.pdf.

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

39 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

40 Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir and Gad M. Landau, editors, *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001. doi:10.1007/3-540-48194-X_17.

41 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.

42 Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 28:1–28:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.28.

43 Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, 1966.

44 Chen Li, Bin Wang, and Xiaochun Yang. VGRAM: improving performance of approximate queries on string collections using variable-length grams. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 303–314. ACM, 2007. URL: http://www.vldb.org/conf/2007/papers/research/p303-li.pdf.

45 Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, March 2016. doi:10.1093/bioinformatics/btw152.

46 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinform.*, 34(18):3094–3100, 2018. doi:10.1093/bioinformatics/bty191.

47 Veli Mäkinen and Gonzalo Navarro. Position-restricted substring searching. In José R. Correa, Alejandro Hevia, and Marcos A. Kiwi, editors, *LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings*, volume 3887 of *Lecture Notes in Computer Science*, pages 703–714. Springer, 2006. doi:10.1007/11682462_64.

48 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

49 Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

50 Guillaume Marçais, Dan F. DeBlasio, and Carl Kingsford. Asymptotically optimal minimizers schemes. *Bioinform.*, 34(13):i13–i22, 2018. doi:10.1093/bioinformatics/bty258.

51 Guillaume Marçais, David Pellow, Daniel Bork, Yaron Orenstein, Ron Shamir, and Carl Kingsford. Improving the performance of minimizers and winnowing schemes. *Bioinform.*, 33(14):i110–i117, 2017. doi:10.1093/bioinformatics/btx235.

52 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 408–424, 2017. doi:10.1137/1.9781611974782.26.

53 Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Carl Kingsford. Compact universal k-mer hitting sets. In Martin C. Frith and Christian Nørgaard Storm Pedersen, editors, *Algorithms in Bioinformatics - 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22-24, 2016. Proceedings*, volume 9838 of *Lecture Notes in Computer Science*, pages 257–268. Springer, 2016. doi:10.1007/978-3-319-43681-4_21.

54 Jianbin Qin, Wei Wang, Chuan Xiao, Yifei Lu, Xuemin Lin, and Haixun Wang. Asymmetric signature schemes for efficient exact edit similarity query processing. *ACM Trans. Database Syst.*, 38(3):16:1–16:44, 2013. doi:10.1145/2508020.2508023.

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

56 Craige Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961. doi:10.4153/CJM-1961-015-3.

57 Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. Winnowing: Local algorithms for document fingerprinting. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 76–85. ACM, 2003. doi:10.1145/872757.872770.

58 Yihan Sun and Guy E. Blelloch. Parallel range, segment and rectangle queries with augmented maps. In Stephen G. Kobourov and Henning Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*, pages 159–173. SIAM, 2019. doi:10.1137/1.9781611975499.13.

59 The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 5.2.1 edition, 2021. URL: https://doc.cgal.org/5.2.1/Manual/packages.html.

60 Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 85–96. ACM, 2012. doi:10.1145/2213836.2213847.

61 Xiaoli Wang, Xiaofeng Ding, Anthony K. H. Tung, and Zhenjie Zhang. Efficient and effective KNN sequence search with approximate n-grams. *Proc. VLDB Endow.*, 7(1):1–12, 2013. doi:10.14778/2732219.2732220.

62 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

63 Derrick E. Wood and Steven L. Salzberg. Kraken: Ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3), 2014. Copyright: Copyright 2014 Elsevier B.V., All rights reserved. `doi:10.1186/gb-2014-15-3-r46`.

64 Zhenglu Yang, Jianjun Yu, and Masaru Kitsuregawa. Fast algorithms for top-k approximate string matching. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1939`.

65 Minghe Yu, Jin Wang, Guoliang Li, Yong Zhang, Dong Deng, and Jianhua Feng. A unified framework for string similarity search with edit-distance constraint. *VLDB J.*, 26(2):249–274, 2017. `doi:10.1007/s00778-016-0449-y`.

66 Haoyu Zhang and Qin Zhang. Minsearch: An efficient algorithm for similarity search under edit distance. In Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash, editors, *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 566–576. ACM, 2020. `doi:10.1145/3394486.3403099`.

67 Zhenjie Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, and Divesh Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 915–926. ACM, 2010. `doi:10.1145/1807167.1807266`.

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

69 Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Lower density selection schemes via small universal hitting sets with short remaining path length. In Russell Schwartz, editor, *Research in Computational Molecular Biology - 24th Annual International Conference, RECOMB 2020, Padua, Italy, May 10-13, 2020, Proceedings*, volume 12074 of *Lecture Notes in Computer Science*, pages 202–217. Springer, 2020. `doi:10.1007/978-3-030-45257-5_13`.

70 Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Sequence-specific minimizers via polar sets. *bioRxiv*, 2021. `doi:10.1101/2021.02.01.429246`.