The anti-lexicographic SUS-anchor: a near-optimal k=1 sampling scheme

Ragnar Groot Koerkamp
Karlsruhe Institute of Technology, Germany

Abstract

Motivation. In recent years, there has been a renewed interest in the search for low density minimizer schemes. These schemes take a *window* of w consecutive k-mers, and *sample* one of them: the smallest under some specific order. Schemes such as the mod-minimizer provide a low *density* (fraction of sampled k-mers) when $k \gg w$, while schemes such as the greedy minimizer work well for explicit small parameters roughly in the regime $k \leq 2w$, for k and w up to 15 or so.

When $k < \log_\sigma w$ is very small, minimizer schemes cannot do well, and more general *sampling* schemes are needed that can be richer than just comparing k-mers. Bidirectional-string anchors (bd-anchors) form one such scheme.

Methods. Inspired by bd-anchors, we introduce the *smallest unique substring* or SUS-anchor: Given a window, this considers all suffixes that do not occur as a substring elsewhere in the window. It then samples the start position of the smallest suffix according to the new *anti-lexicographic* order that minimizes the first character and maximizes the remaining characters. We give a linear-time and $O(w)$ space streaming algorithm to compute all SUS-anchors of a string.

Results. For alphabet size $\sigma = 4$ and $k = 1$, the anti-lexicographic SUS-anchor empirically has density $< 1\%$ away from the density lower bound, significantly improving over bd-anchors that are often $> 15\%$ above it. For alphabet size $\sigma = 2$, the density is at most 10% above the lower bound, which again improves over the $> 50\%$ overhead of bd-anchors.

2012 ACM Subject Classification Theory of computation → Sketching and sampling; Applied computing → Bioinformatics

Keywords and phrases Minimizers; Sampling scheme; Sketching; Maximal suffix; Smallest unique substring

Supplementary Material *Software*: `https://github.com/RagnarGrootKoerkamp/minimizers`

1 Introduction

Minimizers [25, 24] are a technique to subsample the k-mers of a string such that consecutive samples are at most w positions apart. Minimizer sampling has many applications in bioinformatics. In particular, the minimizers of a string are a *sketch* or compressed (lossy) representation, allowing for faster processing. Specific applications include seeding for read-mapping as done by minimap [21], the minimizer-space De Bruijn graph [8], text-indexing using the U-index [1], sketching for Jaccard similarity in mashmap [18], host depletion in Deacon [6], and more [30].

Recent work. There is a long line of active work on minimizers. Of particular interest is the *density*, which is the expected fraction of sampled positions. Recently, the mod-minimizer [15, 13] introduced schemes with near-optimal density for $k \to \infty$. The GreedyMini [11] is the state of the art for smaller w and k roughly up to 15, achieving particularly close to optimal density for $k = w+1$, $k = w$, and k just below w. Shur et al. recently introduced *spacers* [27] that prefer sampling k-mers starting with `10` (after projecting to a binary alphabet) for which the next occurrence of `10` is as far ahead as possible. Spacers improve the density of the ABB+ scheme we introduce here, and are especially good when w is large. For particularly

small parameters $(\sigma, k) = (2, \leq 6)$ and $(\sigma, k) = (4, 2)$, exactly optimal *minimizer* schemes are given by the search algorithm OptMini [28] and analysed theoretically by Shur [26].

This raises the question: **can we find near-optimal *sampling* schemes for small k, and in particular for $k = 1$?** An ILP search suggests that the answer is yes: this is always possible when $k \equiv 1 \mod w$ [19], and thus we are motivated to search for a "clean", constant-space scheme.

Some further recent work includes SimdMinimizers [14], a SIMD-based algorithm that computes *random minimizers* at around 500 Mbp/s. A precise analysis of the density of the random minimizer is given in [10]. Vigemers [16] reduce the imbalance when using minimizers for partitioning, and multiminimizers [17] reduce the density at the cost of more compute.

Sampling and selection schemes. Minimizer schemes hash each k-mer and sample the k-mer with the smallest hash in a window of w k-mers (or $\ell = w + k - 1$ characters). Minimizers cannot achieve a good density of $O(1/w)$ for constant k as $w \to \infty$ [23], but this restriction does not apply to *sampling schemes*, which have more freedom because they are not required to first hash all k-mers. In particular, *selection schemes* with $k = 1$ simply sample a single *position* based on a window of length $w = \ell$. Bidirectional string anchors (bd-anchors) are one such selection scheme [22]. These sample the start position of the smallest *rotation* of a window and achieve a density of $O(1/w)$. For large w and alphabet size $\sigma = 4$, bd-anchors have a density around 20% above the lower bound. Our goal is to reduce this to 0% overhead.

Contributions. In this paper, we introduce SUS-anchors, short for *smallest unique substring* anchors. As the name implies, given a window, these sample the start position of the smallest substring that does not occur a second time. This is equivalent (after reversing the order of the alphabet) to finding the *maximal suffix* of each window. Specifically:

- We introduce the concept of *character-based orders* that generalizes e.g. the lexicographic, alternating [24], and ABB [9] orders.
- We then introduce two new orders: the *ABB+* and *anti-lexicographic* order.
- We give an $O(n)$ time and $O(\ell)$ space sliding-window algorithm to compute the SUS-anchors of a string of length n that is based on the *monotone-queue* approach [14].
- We experimentally show that SUS-anchors with the anti-lexicographic order have density within 1% of optimal for $\sigma = 4$ and any w.

Parts of these results have been previously introduced in the author's PhD thesis [12].

2 Preliminaries

2.1 Minimizers and sampling schemes

For a more in-depth introduction to minimizers, we refer the reader to the survey by Zheng et al. [30], the mod-minimizer paper [15], and the author's thesis [12]. Here we briefly introduce the required notation and concepts.

Notation. For an integer w, we write $[w] = \{0, 1, \ldots, w - 1\}$, and for a string $W = w_0 \ldots w_{|W|-1}$ we use $W[i \ldots j)$ to indicate the substring $w_i \ldots w_{j-1}$. We assume an alphabet Σ of size σ.

Sampling schemes. Given parameters $w \geq 1$ and $k \geq 1$, a *window* is a string over Σ of length $\ell = w + k - 1$ that contains exactly w k-mers. A *local sampling scheme* or just *sampling scheme* is a function $f : \Sigma^\ell \to [w]$ that indicates that from window W, the k-mer $W[f(W) \dots f(W) + k)$ is *sampled*.

Given a text T, we are interested in the *set* of all sampled positions when sliding window W over the text. Consecutive sampled positions differ by at most w, and a sampling scheme is *forward* when the sampled position in T never decreases when sliding the window forward.

Often considered are *minimizer schemes* [25, 24], which are forward sampling schemes that sample the start position of the smallest k-mer (according to some order given by a hash function) starting in the window.

In this paper, we are particularly interested in *selection* schemes, which are sampling schemes with $k = 1$ [29].

Density. The *particular density* of a sampling scheme is the fraction of positions that are sampled. The *density* is the expected value of the particular density on a random string with length going to infinity. For forward schemes, the density can be computed as the probability that two *consecutive* windows W and W' (overlapping by $\ell - 1$ characters and together forming a *context* of $\ell + 1$ characters) sample a different position.

Density lower bounds. A trivial lower bound on the density of sampling schemes is $1/w$, since at least one in every w positions is sampled. The best current lower bound for *forward* sampling schemes [19] simplifies (with a small loss of accuracy) to $\frac{1}{w+k}\left\lceil\frac{w+k}{w}\right\rceil$. For $k = 1$, this gives a lower bound for forward selection schemes of $\frac{1}{w+1}\left\lceil\frac{w+1}{w}\right\rceil = \frac{2}{w+1}$. The main insight is that the density equals the expected number of samples in a random cyclic string (cycle) of length $w + 1$, and in any such cycle, at least two different positions must be sampled.

For *minimizer* schemes, a lower bound is given by $1/\sigma^k$ [23]: if $w \to \infty$, exactly all occurrences of the smallest k-mer will be sampled, and these occur every σ^k positions in expectation. Note, however, that this bound does *not* hold for general sampling schemes, and it is exactly this property that will allow us to design near-optimal sampling schemes for small k.

2.2 Bidirectional string anchors

Bidirectional string anchors (*bd-anchors*) are a $k = 1$ selection scheme introduced by Loukides, Pissis, and Sweering for the purpose of text indexing [22]. Given a window, they sample the (leftmost) start position of the lexicographically smallest rotation. A drawback of bd-anchors is that they are not forward: the window `ZABAAC` has `AAC...` as smallest rotation, while the shifted window `ABAACA` has `AAB...` as smallest rotation. After further shifting to `BAACAY`, the smallest rotation is `AAC...` again: the `AB` prefix caused the selected position to jump around. To ensure the density[1] is in $O(1/w)$, *reduced* bd-anchors [22] avoid sampling the last $r = C\lceil\log_\sigma \ell\rceil$ positions, where $C = 4$ in theory but in practice $C = 3$ or less suffices. Given the choice of $r > \log_\sigma \ell$, in most windows of a random string the bd-anchor is the same as the smallest lexicographic $(r + 1)$-mer. Note though that reduced bd-anchors are still not forward.

[1] Note that even though typically $\ell = w + k - 1 = w$, we will w for the density and ℓ for the length of the window.

In practice, bd-anchors can be computed in $O(n\ell)$ time by using Booth's linear-time algorithm for the lexicographically minimal rotation [5]. An $O(n)$ time algorithm is possible using a data structure of Kociumaka [20], but this is mostly of theoretical interest only since it requires $O(n)$ words of space for e.g. a suffix array. Theorem 3 of [22] gives an $O(n)$ time and $O(\ell)$ space algorithm by using Kociumaka's method on overlapping chunks of size e.g. 2ℓ.

2.3 Maximal suffixes

Separate from the literature on minimizers, there is a line of work on the *non-empty minimal suffix* and *maximal suffix* of a string [7]. While these two problems appear similar, as one might simply reverse the order of the alphabet, a crucial point is that a string A is always smaller than B when A is a prefix of B. Thus, the minimal suffix prefers shorter suffixes, while the maximal suffix prefers longer suffixes. Because of this, minimal suffixes are less stable as we slide a window over a text, and we do not further consider them.

Babenko et al. [3] introduce an algorithm that preprocesses a string of length n into a linear-space data structure and can then find the maximal suffix of query substrings of length ℓ in $O(\log \ell)$ time, which was improved to $O(1)$ query time in [2].

3 Character-based orders

To capture some of the existing "lexicographic-like" orders, we define *character-based orders* that compare strings one character at a time[2]. This is a natural requirement in our setting, because it allows comparing suffixes of a string without worrying that future characters (after shifting the window) will change the order, as long as one is not a prefix of the other.

▶ **Definition 1** (Character-based order). *An order O on strings over Σ is* character-based *if there exist orders O_i on Σ for $i \in \{0, 1, 2, \dots\}$ such that for all strings $A = a_0 \dots a_{|A|-1}$ and $B = b_0 \dots b_{|B|-1}$ with longest common prefix $\ell = \mathsf{lcp}(A, B)$ we have*

$$A <_O B \qquad \text{iff} \qquad (A \text{ is a strict prefix of } B) \text{ or } a_\ell <_{O_\ell} b_\ell.$$

The orders can be either total orders*, or* linear preorders *when equalities are allowed.*

This scheme encapsulates the **lexicographic order** on strings, where each O_i is simply the lexicographic order on Σ. The main drawback of the lexicographic order is that it clusters small strings: since `AAAAX` is small, the next k-mer `AAAXY` is also small, possibly causing consecutive positions to be sampled as minimizers. Most of the following schemes instead look for a *transition* from a small character (`A`) to a large (`Z`) or non-small (`BCD..Z`) character.

The **alternating order** [24] uses the lexicographic order for even i, and the reverse lexicographic order for odd i, so that `AZAZAZ...` is the smallest string. The **ABB order** [4, 9] uses the lexicographic order for O_0, and for $i > 0$ it uses the order

$$1 =_{O_i} 2 =_{O_i} \cdots =_{O_i} \sigma - 1 <_{O_i} 0,$$

so that any string like `ABBBB...` or `AXYZDEF...` is minimal. This scheme has the nice property that occurrences of small strings starting in `A` and not containing further `A`'s are

[2] A slight generalization of this concept that we do not otherwise need in this paper uses orders O_i on strings of length i and then compares *prefixes* $a_0 \dots a_\ell <_{O_\ell} b_0 \dots b_\ell$ instead.

disjoint [4]. **Vigemers** [16] are also a character-based order, where O_i is the order after xor'ing by a character γ_i.

A drawback of the ABB order is that it throws away some information: for example, over the normal alphabet, `AB` and `AC` are considered equal. Thus, we also consider a version with tiebreaking, *ABB+*:

▶ **Definition 2** (ABB+ order). *The* ABB+ *order first compares two strings via the ABB order, and, in case of a tie, compares them via the plain lexicographic order.*

Note that this is only useful when comparing strings (k-mers) of equal length, and that the ABB+ order is not itself a character-based order.

The following scheme is more practical.

▶ **Definition 3** (Anti-lexicographic order). *The* anti-lexicographic order *uses the lexicographic order for O_0, and* reverse *lexicographic order for O_i for $i > 0$.*

In this order, the smallest alphabetic string is `AZZZZ...`.

4 Smallest unique substring anchors

Intuition. Consider again the bd-anchor, which samples the start position of the smallest rotation of a window W. As we saw in Section 2.2, a drawback is that rotations "wrap around", and that the character $W[0]$ at the start of a string influences whether the rotation starting at the last character $W[\ell-1]$ is small or not. This is easily fixed by considering only the smallest *suffix* instead. However, the smallest suffix is the empty suffix, and so we could sample the first character of the smallest *non-empty* suffix. Unfortunately, this results in a bad density: a random window ends in the smallest symbol 0 with probability $1/\sigma$, in which case the suffix consisting of just this character is the smallest one, and the density will be around $1/\sigma$ regardless of w. To avoid this, we impose the following restriction (which, in a way, generalizes the "non-empty" condition): a suffix is only allowed to be sampled if it does not occur elsewhere in the window as a substring. Thus, we look for the *smallest unique suffix* $W[i\dots)$. Let $S = W[i\dots j)$ be the shortest prefix of $W[i\dots)$ that is unique in W. Then S is the *smallest unique substring* (SUS[3]) of W [12]. As an example, the string `CABBAB` has `AB` as its smallest suffix, and `ABBAB` as its smallest *unique* suffix. The smallest unique *substring* is `ABB`, and the index of the correspondingly sampled *SUS-anchor* is 1.

▶ **Definition 4** (SUS-anchor). *Given a window W of length $w = \ell$, the* smallest unique substring *is the smallest substring* $\mathsf{SUS}(W) = W[i\dots j)$ *that does not occur elsewhere in W. Then $W[i\dots\ell)$ is the smallest unique* suffix*, and the SUS-anchor is i.*

MS-anchor. It turns out that the concept of *smallest unique suffix* is exactly equivalent to that of the *maximal suffix* [7, 2] after reversing the order of the alphabet: in both cases, we look for the extremal suffix where longer suffixes should be preferred over shorter ones.

▶ **Definition 5** (MS-anchor). *Given a window W of length $w = \ell$, the* maximal suffix anchor *or MS-anchor samples the position $i \in [w]$ where the maximal suffix $W[i\dots\ell)$ of W starts.*

[3] Not to be confused with the *shortest* unique substring, which is also commonly abbreviated as SUS.

Variants. Alongside the lexicographic variant, the SUS-anchor allows variants based on character-based orders. Specifically, we consider SUS-anchors with the anti-lexicographic order.

These variants work just like the lexicographic version: simply consider the set of suffixes that do not occur as a substring elsewhere, and then take the smallest of these using the chosen character-based order.

4.1 Properties

We now state some observations and then prove some properties of SUS-anchors.

▶ **Observation 6.** *Given a window W, the smallest unique suffix is smaller than all longer suffixes, because all longer suffixes must be unique and thus larger than the smallest unique suffix.*

▶ **Observation 7.** *Removing the last character from the smallest unique substring results in a non-unique substring.*

▶ **Theorem 8** (SUS-anchors are forward)**.** *SUS-anchors are forward for any character-based order.*

Proof. Consider two consecutive windows W and W' with $\mathsf{SUS}(W) = W[i \dots j)$. If $i = 0$, the scheme is trivially forward. Otherwise, let $0 \leq i' < i - 1$ be the start index of a suffix $W'[i' \dots)$. By the previous observation, $W[i \dots) <_O W[i' + 1 \dots)$, and since $W[i \dots)$ is not a prefix of $W[i' + 1 \dots)$ (for otherwise it would not be unique), appending $W'[\ell - 1]$ to these two suffixes will not change their relative order. Since $W[i \dots)$ is unique in W, $W'[i - 1 \dots)$ is also unique in W', and so $W'[i - 1 \dots)$ is a smaller unique suffix than $W'[i' \dots)$ for all $i' < i - 1$. Thus, $\mathsf{SUS}(W')$ cannot start at an index less than $i - 1$, and thus the sus-anchor is forward. ◀

▶ **Theorem 9** (Charged context)**.** *Given two consecutive windows W and W' that together form a* context*, the sampled SUS-anchor changes if and only if either $\mathsf{SUS}(W)$ is a prefix of W or $\mathsf{SUS}(W')$ is a suffix of W'. In this case, the context is* charged*.*

Proof. Let $\mathsf{SUS}(W) = W[i \dots j)$ and $\mathsf{SUS}(W') = W'[i' \dots j')$.

If $i = 0$ is sampled, W' cannot sample the same position. If $W'[i' \dots j') = W[i' \dots \ell)$ is a suffix, then $W'[i' \dots j' - 1)$ is not unique in W', and thus also not in W. Thus, W must sample a different position.

For the other direction, assume that $i' > i - 1$, so that different text positions are sampled. If $i = 0$ or $i' = \ell - 1$ we are done, so assume $i > 0$ and $i' < \ell - 1$.

If $W[i \dots) < W[i' + 1 \dots)$, then for every possible character $W'[\ell - 1]$ that can be appended, we have $W'[i - 1 \dots) < W'[i' \dots)$, which is in contradiction with the fact that $W'[i' \dots)$ is smaller than all longer suffixes. Thus, $W[i' + 1 \dots)$ must be a prefix of $W[i \dots)$, so that the suffix is not unique. This means that $W'[i' \dots \ell - 1)$ is not unique in W', and thus, the SUS $W'[i' \dots j')$ can only end in $j' = \ell$, as required. ◀

4.2 Computing SUS-anchors

Via the equivalence to maximal suffixes, we can use the theoretical algorithm of [2] that requires $O(n)$ space, $O(n)$ preprocessing, and supports $O(1)$ queries for substring suffix-maximum. To reduce the space usage, we can use the same trick as for bd-anchors, and run it on chunks of size 2ℓ that overlap by $\ell - 1$, so that the total space usage is $O(\ell)$ and the total time remains $O(n)$.

▶ **Theorem 10.** *The SUS-anchors of a string of length n can be computed in $O(n)$ time and $O(\ell)$ space.*

Below, we present two direct methods for computing the *maximal* suffix, as this slightly simplifies the notation and matches [2]. It trivially extends to other character-based orders such as the anti-lexicographic order.

A streaming algorithm. We now directly adapt the algorithm of [2] to our streaming setting. Suppose that we have so far processed $T[0 \dots t)$. The algorithm maintains a doubly linked list $A = (a_0, a_1, a_2, \dots)$ of *active* text positions a_i such that $T[a_i \dots t) > T[j \dots t)$ for all $a_i < j < t$. This list is *decreasing* in the sense that $T[a_0 \dots t) > T[a_1 \dots t) > T[a_2 \dots t) > \dots$, and it plays a similar role to the *monotone queue* in the classic minimizer algorithm [14]. Unlike that original case, here we also store a list of *events* for every text position that can modify the middle of the list.

As we shift the window and increment t to $t' = t + 1$, we remove a_0 when $a_0 < t' - \ell$. After appending $T[t]$, we get a new suffix $T[t \dots t')$, and we have to update A.

- **Repeatedly pop A.** We remove the last element $\mathsf{last}(A) = A_{|A|-1}$ from A as long as $T[t \dots t') > T[\mathsf{last}(A) \dots t')$.
- **Add event.** Then, if A is not empty and $T[t \dots) > T[\mathsf{last}(A) \dots)$ (taking into account upcoming characters), compute $\lambda = \mathsf{LCP}(T[t \dots), T[\mathsf{last}(A) \dots))$ and add the event "check if we should drop the element preceding t from A" to the event queue for position $t + \lambda + 1$.
- **Push t.** Then, push t to A and increment t'.
- **Handle event.** As soon as t'' reaches position $t + \lambda + 1$, we check if t is still in A, and if so, if it has now become larger than its preceding element. Then repeatedly remove the preceding element as long as $T[t \dots t'')$ is larger, and end when either there is no preceding element, or when $T[a_i \dots t'') > T[t \dots t'')$. In that case, again check if the order will flip in the future, and if so, add a new event to the event queue.

Practical considerations. To make this practical, all LCP computations can be bounded to ℓ, and the event queues can be stored in a ring buffer with just $\ell + 1$ slots for the current and next ℓ positions. Originally, the LCP computations are done in constant time using the suffix array. Instead, we simply do a word-by-word scan on the bit-representation of the text, which takes expected constant time on random strings since LCPs of, say, over 64 bits are unlikely in random strings.

A more direct algorithm. A final modification drops the event queues and allows us to replace the doubly linked list A by a simple double-ended queue. For each $a \in A$, we now additionally store the text position x_a where a becomes "hot", i.e., we store that a becomes the first element of A (and thus the start of the maximal suffix of the window) when the character $T[x_a]$ enters the sliding window. When comparing t to $\mathsf{last}(A)$, there are three cases.

1. $T[t \dots) < T[\mathsf{last}(A) \dots)$, in which case we push t and note that it becomes hot when $\mathsf{last}(A)$ falls out of the window, i.e., $x_t = \mathsf{last}(A) + \ell$.
2. $T[t \dots) > T[\mathsf{last}(A) \dots)$: t "takes over" $\mathsf{last}(A)$ when character $t + \lambda$ enters the window, where $\lambda = \mathsf{LCP}(T[t \dots), T[\mathsf{last}(A) \dots))$.
   - If $\mathsf{last}(A)$ is already hot by that time ($x_{\mathsf{last}(A)} < t + \lambda$), simply push t.
   - If not, $\mathsf{last}(A)$ will never be hot, and we pop it and (repeatedly) compare t to the new $\mathsf{last}(A)$.

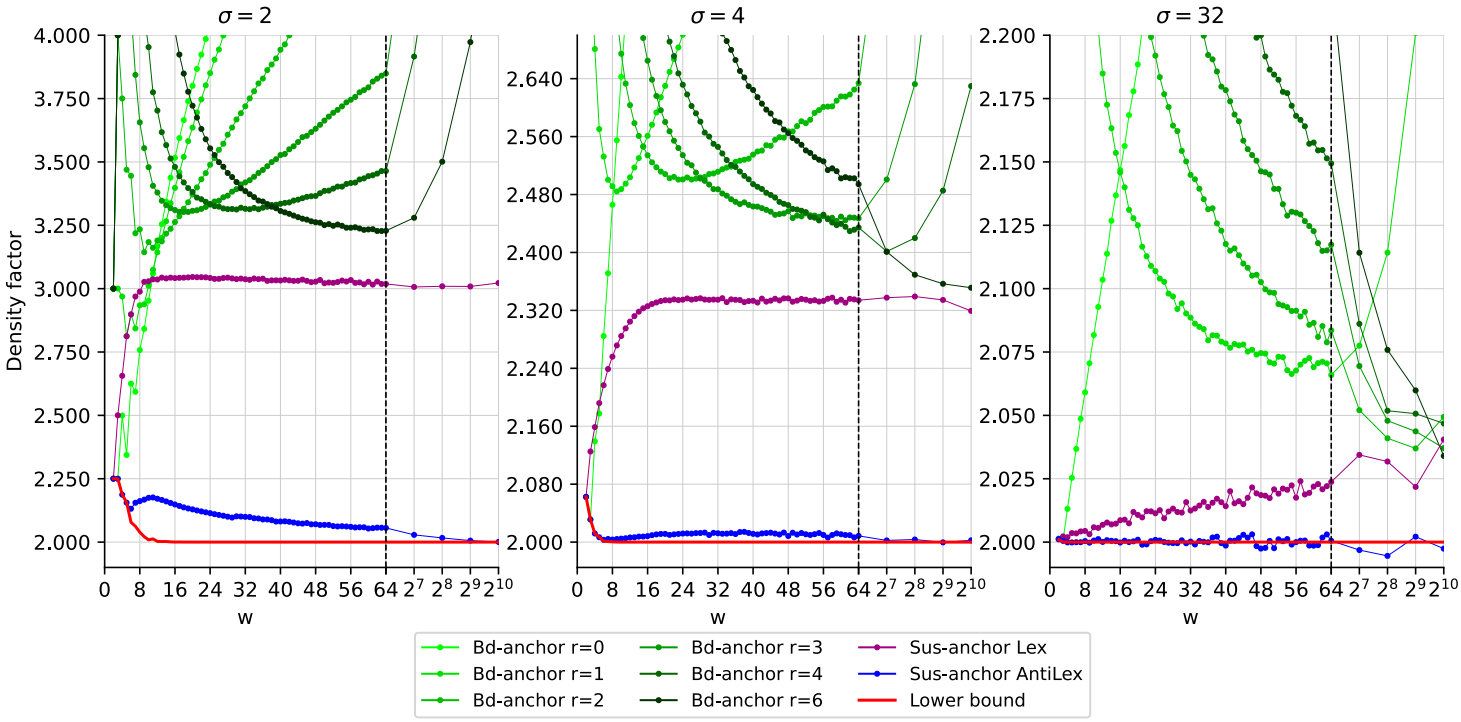

Figure 1 Comparison of the density factor (the density multiplied by $w+1$) of selection schemes for $\sigma \in \{2, 4, 32\}$, $k = 1$, and varying w. The lower bound is shown in red. Bd-anchors (lime/green) are shown for various r, and SUS-anchors are shown with both the lexicographic (purple) and anti-lexicographic order (blue). (The other orders are worse than anti-lexicographic.) On the right of each plot, we show scaling for large w. Each data point is the particular density on an independent random string of length 10^7. For large σ and w, there remains some variance in the estimated density.

Once this process finishes, the first element of A indicates the maximum suffix of the current window. We then increment t to the next text position and check whether we reached the position x_{a_1} where the next element of A becomes hot (either because a_0 falls out of the window or a_1 introduces a new maximal suffix). In that case, we pop a_0.

5 Results

We compare bd-anchors with various r and sus-anchors with various underlying orderings in Figure 1. We see that as w grows, so does the optimal value of r. Still, even with the best choice of r, bd-anchors have density over 50% ($\sigma = 2$), 15% ($\sigma = 4$), or 2.5% ($\sigma = 32$) above the lower bound for all but the smallest w.

Lexicographic SUS-anchors perform consistently slightly better than the best bd-anchor for all σ and not too small w. Anti-lexicographic SUS-anchors are much better, and get surprisingly close to the lower bound. They are away from the lower bound by less than 10% for $\sigma = 2$ and less than 1% for $\sigma = 4$ and $\sigma = 32$.

6 Discussion

We introduced anti-lexicographic SUS-anchors, gave a linear-time algorithm to compute them, and showed that they empirically get to within 1% of the density lower bound for $\sigma = 4$.

Future work. On a practical level, future work is needed to improve the current monotone-queue-based algorithm to a rescan-like approach [14] or even a branchless variant that could be implemented in parallel using SIMD.

It would also be interesting to use SUS-anchors in combination with the mod-minimizer [15], especially in order to remove the mod-minimizer's r parameter, but this has not worked out so far.

Further work is needed to prove that, for example, anti-lexicographic SUS anchors have a density of $2/(w+1) + o(1/w)$. Given how close to optimal these schemes already are, the fact that exactly optimal schemes *do* exist for small w [19], as well as manually constructed schemes that are exactly optimal for $\sigma = 2$ and $w \leq 12$, the question remains whether "clean", constant space schemes can be developed for generic w. The *spacers* of [27] are similar to our own ongoing work, and provide a first step in this direction.

Lastly, the SUS-anchor is a forward scheme. It is known that non-forward schemes can be (at least) slightly better at times and break the forward lower bound, but this is not well understood.

References

1 Lorraine A. K. Ayad, Gabriele Fici, Ragnar Groot Koerkamp, Grigorios Loukides, Rob Patro, Giulio Ermanno Pibiri, and Solon P. Pissis. U-Index: A Universal Indexing Framework for Matching Long Patterns. In *SEA 2025*, volume 338 of *LIPIcs*, pages 4:1–4:18, 2025. doi:10.4230/LIPIcs.SEA.2025.4.

2 Maxim Babenko, Paweł Gawrychowski, Tomasz Kociumaka, Ignat Kolesnichenko, and Tatiana Starikovskaya. Computing minimal and maximal suffixes of a substring. *Theoretical Computer Science*, 638:112–121, 2016. Pattern Matching, Text Data Structures and Compression. doi:10.1016/j.tcs.2015.08.023.

3 Maxim Babenko, Ignat Kolesnichenko, and Tatiana Starikovskaya. On minimal and maximal suffixes of a substring. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching*, pages 28–37, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

4 Simon R. Blackburn. Non-overlapping codes. *IEEE Transactions on Information Theory*, 61(9):4890–4894, September 2015. doi:10.1109/tit.2015.2456634.

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

6 Bede Constantinides, John Lees, and Derrick W Crook. Deacon: fast sequence filtering and contaminant depletion. June 2025. doi:10.1101/2025.06.09.658732.

7 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.

8 Barış Ekim, Bonnie Berger, and Rayan Chikhi. Minimizer-space De Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. *Cell Systems*, 12(10):958–968.e6, October 2021. doi:10.1016/j.cels.2021.08.009.

9 Martin C Frith, Laurent Noé, and Gregory Kucherov. Minimally overlapping words for sequence similarity search. *Bioinformatics*, 36(22–23):5344–5350, December 2020. doi:10.1093/bioinformatics/btaa1054.

10 Shay Golan and Arseny M. Shur. Expected density of random minimizers. In *SOFSEM 2025: Theory and Practice of Computer Science*, page 347–360. Springer Nature Switzerland, 2025. doi:10.1007/978-3-031-82670-2_25.

11 Shay Golan, Ido Tziony, Matan Kraus, Yaron Orenstein, and Arseny Shur. GreedyMini: generating low-density DNA minimizers. *Bioinformatics*, 41:275–284, July 2025. doi:10.1093/bioinformatics/btaf251.

12 Ragnar Groot Koerkamp. *Optimal Throughput Bioinformatics*. PhD thesis, ETH Zurich, 2025. doi:10.3929/ETHZ-C-000783091.

13 Ragnar Groot Koerkamp, Daniel Liu, and Giulio Ermanno Pibiri. The open-closed mod-minimizer algorithm. *Algorithms for Molecular Biology*, 20(4), mar 2025. doi:10.1186/s13015-025-00270-0.

14 Ragnar Groot Koerkamp and Igor Martayan. SimdMinimizers: Computing Random Minimizers, fast. In *SEA 2025*, volume 338 of *LIPIcs*, pages 20:1–20:19, 2025. doi:10.4230/LIPIcs.SEA.2025.20.

15 Ragnar Groot Koerkamp and Giulio Ermanno Pibiri. The mod-minimizer: A simple and efficient sampling algorithm for long k-mers. In *WABI 2024*, volume 312 of *LIPIcs*, pages 11:1–11:23, 2024. doi:10.4230/LIPIcs.WABI.2024.11.

16 Florian Ingels, Antoine Limasset, Camille Marchet, and Mikaël Salson. Vigemers: on the number of k-mers sharing the same xor-based minimizer. *arXiv*, 2026. doi:10.48550/ARXIV.2602.03337.

17 Florian Ingels, Lucas Robidou, Igor Martayan, Camille Marchet, and Antoine Limasset. Minimizer density revisited: Models and multiminimizers. November 2025. doi:10.1101/2025.11.21.689688.

18 Chirag Jain, Alexander Dilthey, Sergey Koren, Srinivas Aluru, and Adam M. Phillippy. A fast approximate algorithm for mapping long reads to large reference databases. *Journal of Computational Biology*, 25(7):766–779, July 2018. doi:10.1089/cmb.2018.0036.

19 Bryce Kille, Ragnar Groot Koerkamp, Drake McAdams, Alan Liu, and Todd J Treangen. A near-tight lower bound on the density of forward sampling schemes. *Bioinformatics*, December 2024. doi:10.1093/bioinformatics/btae736.

20 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)*, volume 54 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:12, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2016.28.

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

22 Grigorios Loukides, Solon P. Pissis, and Michelle Sweering. Bidirectional string anchors for improved text indexing and top-k similarity search. *IEEE Transactions on Knowledge and Data Engineering*, 35(11):11093–11111, November 2023. doi:10.1109/tkde.2022.3231780.

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

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

25 Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD/PODS03. ACM, June 2003. doi:10.1145/872757.872770.

26 Arseny Shur. On minimizers of minimum density. *arXiv*, 2025. doi:10.48550/ARXIV.2506.05277.

27 Arseny Shur, Ido Tziony, and Yaron Orenstein. 10-minimizers: a promising class of constant-space minimizers. *bioRxiv*, 2026. doi:10.64898/2026.03.16.712052.

28 Arseny Shur, Ido Tziony, and Yaron Orenstein. Generating minimum-density minimizers. January 2026. doi:10.64898/2026.01.25.701585.

29 Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Lower density selection schemes via small universal hitting sets with short remaining path length. *Journal of Computational Biology*, 28(4):395–409, April 2021. doi:10.1089/cmb.2020.0432.

30 Hongyu Zheng, Guillaume Marçais, and Carl Kingsford. Creating and using minimizer sketches in computational genomics. *Journal of Computational Biology*, 30(12):1251–1276, 2023. doi:10.1089/cmb.2023.0094.