The Anti-Lexicographic SUS-Anchor: An Empirically Optimal Selection Scheme

Ragnar Groot Koerkamp
Karlsruhe Institute of Technology, Germany

Abstract

Motivation. *Selection schemes* provide a way to select a subset of positions in a text in such a way that no two consecutive selected positions are more than w apart. These selected positions can be used as "anchor" points for text indices such that every sufficiently long pattern corresponds to at least one anchor [2]. Closely related are *sampling schemes*, that sample a *k-mer* from each window of w consecutive k-mers in a text, and the more restricted *minimizer schemes*, that achieve this by taking the smallest k-mer according to some order. In recent years, there has been a renewed interest in the search for *low density* schemes that select/sample only a small fraction of positions/k-mers.

The mod-minimizer [18] provides a near-optimal density of $1/w$ as $k/w \to \infty$, 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 small, minimizer schemes cannot do well [27]. As a first step towards low density sampling schemes in this regime, we fix $k = 1$ and search for a near-optimal selection scheme to improve the existing bidirectional string anchors (bd-anchors) [26, 2].

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 parameter-free anti-lexicographic SUS-anchor empirically has density $< 1\%$ away from the density lower bound and is at least $4\times$ closer to the lower bound than all other tested schemes. For alphabet size $\sigma = 2$, the density is at most 10% above the lower bound, which still improves $2\times$ to $3\times$ the overhead of the random minimizer and is consistently better than the greedy minimizer. Likewise, the anti-lexicographic minimizer performs better than all other schemes apart from the greedy minimizer.

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

Digital Object Identifier 10.4230/LIPIcs.WABI.2026.22

Supplementary Material *Software (Source Code)*: `https://github.com/RagnarGrootKoerkamp/minimizers` [14], archived at `swh:1:dir:6b775d1315385c94ecbd6058cfbba2476fbeb0b4`

Acknowledgements I thank the anonymous reviewers for their helpful feedback.

1 Introduction

Minimizers [36, 35] 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, a number of data structures exploit this *window guarantee* and use minimizers as a locality-sensitive hash: SSHash [32, 33] and *k*ache-hash [22] use the minimizer of a string as the key in a hash map, while the U-index [1] is a text index that considers the minimizers of a pattern. In a similar way, the text index of Loukides et

al. and Ayad et al. [26, 2] works for patterns of length at least ℓ and selects *positions* in the text (*anchors*) that are at most $w = \ell$ apart in a locally consistent way, so that each queried pattern of length $\geq \ell$ can be found via its anchor. Improving their bidirectional string anchors is the main goal of this work, as this directly leads to a smaller text index.

From another point of view, 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 [25], the minimizer-space De Bruijn graph [10], sketching for Jaccard similarity in mashmap [21], host depletion in Deacon [7], and more [41]. In this direction, there are also other schemes such as syncmers [9] and universal hitting sets [29, 30, 28]. In these schemes, each k-mer is sampled independently from its context, which may loosen the window guarantee and does not directly give a suitable key or anchor for each window.

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 [18, 16] introduced schemes with near-optimal density for $k \to \infty$. The GreedyMini [13] 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* [38] 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 [39] and analysed theoretically by Shur [37].

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 $k \equiv 1 \mod w$ [23], and thus we are motivated to search for a "clean", constant-space scheme.

Some further recent work includes SimdMinimizers [17], 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 [12]. Vigemers [19] reduce the imbalance when using minimizers for partitioning, and multiminimizers [20] 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 $w \to \infty$ [27], 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 $w = \ell$. The main reason that selection schemes can achieve better density than sampling schemes is that for a fixed sized window, they can also sample the last $k - 1$ positions, and thus achieve a greater distance between consecutive samples. Bidirectional string anchors (bd-anchors) are one such selection scheme [26]. 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 as close to 0% overhead as possible.

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. The *lexicographic* SUS-anchor is equivalent (after reversing the order of the alphabet) to the the *maximal suffix* of each window. Our main results are as follows:

- We experimentally show that SUS-anchors with the anti-lexicographic order (see below) have density within 1% of optimal for $\sigma = 4$ and any w.
- 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 [17].

Smaller contributions that are of possible independent interest are:

- *character-based orders* that generalize e.g. the lexicographic, alternating [35], and ABB [11] order; and
- two new character-based orders: the *ABB+* and *anti-lexicographic* order.

Parts of these results have been previously introduced in the author's PhD thesis [15].

2 Preliminaries

2.1 Minimizer, sampling, and selection schemes

For a more in-depth introduction to minimizers, we refer the reader to the survey by Zheng et al. [41], the mod-minimizer paper [18], and the author's thesis [15]. 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 $W[f(W) \ldots f(W) + k)$ 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* [36, 35], which are forward sampling schemes that sample (the position of) the smallest k-mer in the window according to some order as given by a hash function.

Selection schemes. In this paper, we are particularly interested in *selection* schemes, which are sampling schemes with $k = 1$ [40]. In particular, these select a *position* rather than *k-mer* in each window. Given a fixed window (context) of ℓ characters, the $w = \ell$ selection scheme has greater freedom than an $w = \ell - k + 1$ sampling scheme, and thus can achieve lower density.

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 [23] 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 $1/\sigma^k$ [27]: if $w \to \infty$, exactly all occurrences of the smallest k-mer will be sampled, and these occur every σ^k positions in expectation. However, 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 [26]. 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 [26] 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 that reduced bd-anchors are still not forward.

In practice, bd-anchors can be computed in $O(n\ell)$ time by using Booth's linear-time algorithm for the lexicographically minimal rotation [6]. An $O(n)$ time algorithm is possible using a data structure of Kociumaka [24], 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 [26] gives an $O(n)$ time and $O(\ell)$ space algorithm when the alphabet size is constant ($\sigma = O(1)$) by using Kociumaka's method on overlapping chunks of size e.g. 2ℓ. A practical linear time algorithm to compute reduced bd-anchors with $r = \lceil 4\log_\sigma w\rceil$ is given by [2]. This works by first computing the lexicographic minimizers of size $k=r$ and only comparing the corresponding rotations in case of ties. Additionally, the *randomized* reduced bd-anchor is introduced that mimics the random minimizer: it considers the smallest r-mer in a window under a pseudo-random order and breaks ties by preferring the lexicographically smallest rotation. In practice, the density of randomized reduced bd-anchors is indistinguishable from the density of random minimizers.

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 [8]. 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. [4] 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 [3].

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

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*

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

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** [35] 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** [5, 11] 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 disjoint [5]. (The concept of sets of minimally overlapping strings has been further explored in [11], but these do not form a character-based order and do not directly work with variable-length strings.) **Vigemers** [19] 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 string is `AZZZZ...`.

[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.

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 [15]. 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.*

We note that for the purposes of the definition, it would be sufficient to only consider the starting position of smallest unique *suffix*. Nevertheless, we prefer to consider the smallest unique *substring* since it has properties that are similar to the k-mers sampled by minimizer schemes. For example, both optimal-density minimizer schemes and optimal-density SUS-anchors would always sample non-overlapping k-mers and SUS-anchors respectively.

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.

MS-anchor. It turns out that the concept of *lexicographic smallest unique suffix* is exactly equivalent to that of the *maximal suffix* [8, 3] 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. More generally, for any character-based order O, the SUS-anchor under O starts at the same position as the maximal suffix when suffixes are compared by the reverse of O. Whereas the SUS-anchor explicitly requires considering a unique suffix, this is handled implicitly by the maximal suffix: when one suffix is a prefix of a longer suffix, the longer suffix is automatically the larger one (regardless of the chosen character-based order O).

▶ **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.

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 compute all lexicographic SUS-anchors using the theoretical algorithm of [3] that requires $O(n)$ space, $O(n)$ preprocessing, and supports $O(1)$ queries for substring suffix-maximum. When the alphabet has constant size, we can use the same trick as for bd-anchors to reduce the space usage: Run the construction algorithm on chunks of size 2ℓ that overlap by $\ell - 1$, so that the total space usage is $O(\ell)$ and the total time remains $O(n)$.

Below, we first adapt the method of [3] for computing the maximal suffix of each window to our streaming setting. Then, we adapt it to compute SUS-anchors for any character-based order.

▶ **Theorem 10.** *For any character-based order O, the SUS-anchors of a string of length n over a constant-size alphabet can be computed in $O(n)$ time and $O(\ell)$ space.*

A streaming algorithm for the maximal suffix. We first adapt the algorithm of [3] 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 [17]. 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 $T[t \dots t')$. We update A before pushing the new element t.

- **Repeatedly pop** A**.** We remove the last element $\mathsf{last}(A) = A_{|A|-1}$ from A as long as $T[\mathsf{last}(A) \dots t') < T[t \dots t')$.
- **Add event.** Then, if A is not empty and $T[\mathsf{last}(A) \dots) < T[t \dots)$ (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 to $t' = t + 1$.
- **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 for SUS-anchors. We now make some modifications to use the algorithm above for SUS-anchors. Most importantly, A will now be an *increasing* list of suffixes, with the smallest at the start, with the caveat that unlike usually a string is considered *smaller* than all its prefixes. A final modification drops the event queues and allows us to replace the doubly linked list A by a simple double-ended queue, as shown in Algorithm 1. 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. A is empty, in which case we push t and set $x_t = t$.
2. $T[\mathsf{last}(A) \dots \mathsf{last}(A) + \ell) \leq_O T[t \dots t + \ell)$, in which case we push t and note that it becomes hot (the smallest suffix) when $\mathsf{last}(A)$ falls out of the window, i.e., $x_t = \mathsf{last}(A) + \ell$.
3. $T[\mathsf{last}(A) \dots \mathsf{last}(A) + \ell) >_O T[t \dots t + \ell)$: 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 ($x_{\mathsf{last}(A)} < t + \lambda$), push t with $x_t = \min(t + \lambda, a + \ell)$.
   - If not, $\mathsf{last}(A)$ will never be hot. We pop it and (repeatedly) compare t to $\mathsf{last}(A)$.

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.

Algorithm 1 Pseudo code for the streaming algorithm to compute the SUS-anchors as given in Section 4.2.

```
1:  procedure SUS-ANCHORS(T, ℓ, O)
2:      A ← []                                   ▷ Deque of (position, hot-time) pairs.
3:      S ← []                                   ▷ List of SUS-anchor positions of each window.
4:      for t ← 0, 1, ..., n − 1 do
5:          x_t ← t                              ▷ Moment where T[t...) becomes hot.
6:          while A is not empty do
7:              (a, x_a) ← last(A)
8:              if T[a...a+ℓ) ≤_O T[t...t+ℓ) then        ▷ New suffix is larger.
9:                  x_t ← a + ℓ                  ▷ Hot when a exits the window.
10:                 break
11:             else                             ▷ New suffix is smaller: takes over a at time x
12:                 λ ← LCP(T[t...t+ℓ), T[a...a+ℓ))
13:                 x ← min(t + λ, a + ℓ)
14:                 if x_a < x then              ▷ a becomes hot before t takes over; push t after a.
15:                     x_t ← x
16:                     break
17:                 else                         ▷ a will never be hot; discard it and continue.
18:                     A.pop_last()
19:         A.push((t, x_t))
20:         (p, x_p) ← A.front()
21:         S.push(p)                            ▷ SUS-anchor of T[t − ℓ + 1...t + 1)
22:         if p + ℓ − 1 = t or (|A| > 1 and A.index(1).x ≤ t + 1) then
23:             A.pop_front()                    ▷ Pop from A if window ends or next element becomes hot.
24:     return S
```

5 Results

In Figure 1, we compare the density of SUS-anchors against the bd-anchors selection scheme as well as minimizer schemes. For the minimizer schemes, we choose the k that minimizes the density, and we adjust w to $w' = w + 1 - k$ accordingly to preserve the overall window length $\ell = w' + k - 1$. The true density is approximated by computing the *particular density* on a uniform random string of 10^7 characters over alphabets of size 2, 4, and 32. A new independent random string is used for each data point.

Lexicographic minimizers are roughly on-par with bd-anchors, and lexicographic SUS-anchors are slightly better but still far from optimal. Double decycling [31] and the random minimizer appear to converge to density factor 2 as $w \to \infty$ and have similar density. The greedy minimizer [13] only performs well for small w and σ, as it requires precomputed orders that are not always available. (We did not implement the logic to extrapolate from smaller orders.) The greedy minimizer is the only scheme that improves over anti-lexicographic minimizers (excluding very small $w \leq 6$).

The anti-lexicographic minimizer consistently improves over the random minimizer and double decycling. The anti-lexicographic SUS-anchors consistently have the best density of all, while also being parameter-free. They 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 $\sigma = 32$. For $\sigma = 32$, the anti-lexicographic sus-anchor is indistinguishable from optimal for all $w \leq 64$, and improves over the random minimizer that is $> 1\%$ away from optimal here.

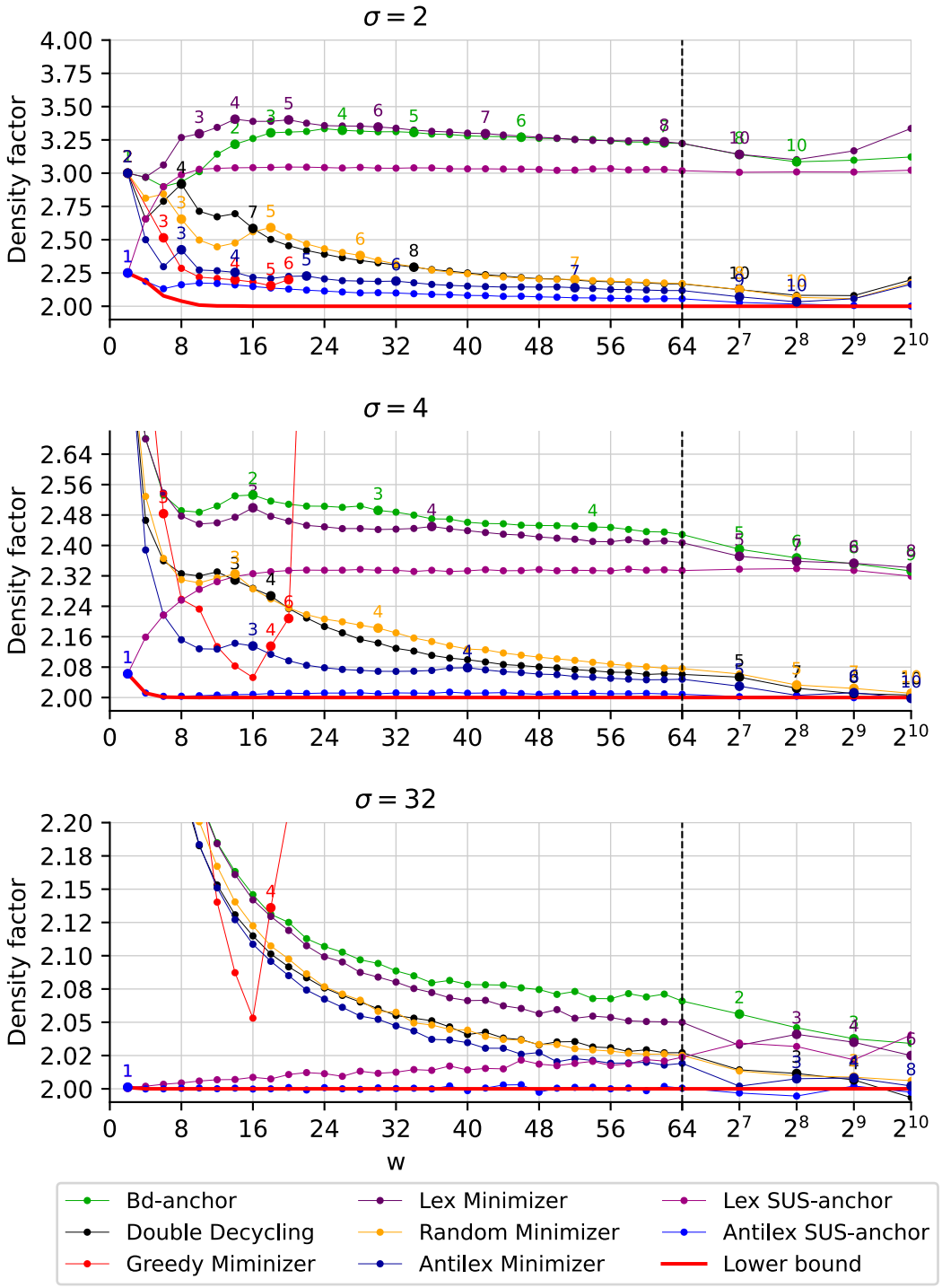

Figure 1 Comparison of the density factor (the density multiplied by $w+1$) of selection schemes for $\sigma \in \{2, 4, 32\}$, and varying w. The lower bound is shown in red. For each scheme (minimizers, bd-anchors [26], double decycling [31], greedy minimizer [13]), the best parameter r or k is chosen for each w (and w is decreased accordingly). Changes to the value of r or k are annotated. 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, occasionally leading to estimates below the lower bound. Sus-anchors are consistently better than minimizers, and the anti-lexicographic order is better than the random order which itself is better than the lexicographic order.

Table 1 Density factor and throughput of various selection schemes for $w = 31$ (left half) and $w = 63$ (right half) on a random string and on chromosome 19. For minimizer schemes, the value of k that minimizes the density on chromosome 19 is chosen. Schemes are sorted by decreasing density.

	$w = 31$	Density		Thrpt.	$w = 63$	Density		Thrpt.
Scheme	k	Random	Ch.19	Mbp/s	k	Random	Ch.19	Mbp/s
Bd-anchor	2	2.59	3.04	2.0	3	2.53	3.36	1.0
Greedy minimizer	4	3.62	2.75	44.9	5	3.68	2.86	43.9
Lex minimizer	4	2.47	2.91	35.3	5	2.42	3.16	35.4
Lex SUS-anchor	-	2.33	2.66	19.0	-	2.34	2.96	18.4
Random minimizer	4	2.18	2.27	40.3	4	2.08	2.19	40.7
Double decycling	5	2.18	2.23	13.6	5	2.10	2.13	13.7
Antilex minimizer	3	2.07	2.29	36.2	4	2.05	2.42	35.0
Antilex SUS-anchor	-	2.01	2.18	17.9	-	2.01	2.33	17.6

Density and throughput on chromosome 19. In Table 1, we compare the density on a random string with the particular density on chromosome 19 of CHM13 [34], which has length 61.7 Mbp. Most schemes have slightly worse density on chromosome 19 than on real data. The greedy minimizer has worse than expected performance in both cases, which is likely due to the fact that our parameters are large and require extrapolating from smaller cases. The anti-lexicographic SUS-anchor has the best performance on random data, but on chromosome 19, it is outperformed by the random minimizer and double decyling [31] for $w = 63$.

Additionally, Table 1 shows the obtained throughput of evaluating each scheme on an Intel i7-10750H CPU running at 2.6 GHz, but we note that the implementations have not been optimized for speed. In particular, for bd-anchors we use a very naive $O(nw^2)$ implementation. Most other methods are $O(n)$ in practice, as can be seen from the fact that they do not slow down from $w = 31$ to $w = 63$. SIMD-minimizers [17], an optimized library for computing random minimizers, has a throughput over 500 MB/s, which is more than 10× faster than all of the methods shown here.

Intuition on SUS-anchor density. Low-density schemes sample positions that are as far apart as possible. When sampling k-mers, this means that ideally each k-mer does not overlap with the preceding or succeeding sampled k-mer, or else as little as possible.[4] Thus, shorter SUS-anchors are beneficial since they are both less likely to overlap and can be the anchor for more consecutive k-mers.

Taken together, this means that short unique substrings should work well. In particular, the anti-lexicographic order is better than the lexicographic order since it avoids sampling overlapping substrings: in the lexicographic order, consecutive substrings like `AAAB` and `AAB` are both small, causing overlapping samples when a window starts with such a string. The anti-lexicographic order prevents this by explicitly preferring a large suffix after removing the first character. A weakness of the ABB scheme is that it needs to sample a relatively

[4] In particular, a minimizer scheme that has density exactly equal to the lower bound [23] must never sample overlapping k-mers: The lower bound uses that at least two k-mers must be sampled on each primitive cycle of $w + k$ characters, and if two overlapping k-mers are sampled, there must be at least a third sampled k-mer. Similarly, SUS-anchors can only have optimal density if they never overlap.

long substring before it becomes unique: in a window like ABDDCBA, 6 of the 7 characters (ABDDCB) are part of the ABB-SUS, so that this sample can only cover a few windows. The anti-lexicographic SUS is simply AB instead.

This also explains the effect of the alphabet size: for $\sigma = 2$, on average a substring needs to have length $\log_2 w$ before it becomes unique. Thus, the SUS-anchors are longer and more likely to overlap. An example of a case where anti-lexicographic SUS-anchors still overlap is given by the consecutive windows ABABABB and BABABBB, with anti-lexicographic SUS-anchors ABABA and ABA. For alphabet size $\sigma = 2$, such cases are somewhat common. For $\sigma = 32$ on the other hand, most characters in the window are unique (for $w \leq 32$) and most anchors will consist of only one or two characters (e.g. just A or AZ) and thus rarely overlap.

6 Conclusion

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 $\sigma = 4$. They are consistently the lowest density selection scheme of all tested variants, while also being parameter-free. As a second best, anti-lexicographic minimizers also perform better than all other schemes apart from the greedy minimizer.

Future work. More work is needed to *prove* that, for example, anti-lexicographic SUS anchors have a close-to-optimal density of $2/(w+1) + o(1/w)$. Given how close to optimal these schemes already are, the fact that *exactly* optimal schemes are known to exist for small w [23], including manually constructed for $\sigma = 2$ and $w \leq 12$, the question remains whether "clean", constant space schemes can be developed for generic w. The *spacers* of [38] are similar to our own ongoing work, and provide a first step in this direction.

On a practical level, further work is needed to improve the current monotone-queue-based algorithm to a rescan-like approach [17] 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 [18]. In particular, the mod-minimizer requires a lower bound $r = \Theta(\log_\sigma w)$ on the length of the chosen minimizers (typically around 4 in practice), and it would be interesting to see if "mod-SUS-anchors" can work without such a restriction. More generally, SUS-anchors can be extended to $k > 1$ by simply excluding the last $k - 1$ suffixes, and preliminary results show that this gives good density up to $k \approx \log_\sigma w$.

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 Lorraine A. K. Ayad, Grigorios Loukides, and Solon P. Pissis. Text indexing for long patterns using locally consistent anchors. *The VLDB Journal*, 34(5), July 2025. doi:10.1007/s00778-025-00935-7.

3 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.

4 Maxim Babenko, Ignat Kolesnichenko, and Tatiana Starikovskaya. On minimal and maximal suffixes of a substring. In *Combinatorial Pattern Matching*, page 28–37. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38905-4_5.

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

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

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

8 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.

9 Robert Edgar. Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences. *PeerJ*, 9:e10805, February 2021. doi:10.7717/peerj.10805.

10 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.

11 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.

12 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.

13 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.

14 Ragnar Groot Koerkamp. RagnarGrootKoerkamp/minimizers. Software, swhId: swh:1:dir:6b775d1315385c94ecbd6058cfbba2476fbeb0b4 (visited on 2026-08-14). URL: https://github.com/RagnarGrootKoerkamp/minimizers, doi:10.4230/artifacts.27679.

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

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

17 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.

18 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.

19 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.

20 Florian Ingels, Lucas Robidou, Igor Martayan, Camille Marchet, and Antoine Limasset. Minimizer density revisited: Models and multiminimizers. In *RECOMB*, November 2026. doi:10.1101/2025.11.21.689688.

21 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.

22 Jamshed Khan, Rob Patro, and Prashant Pandey. $k$ache-hash: A dynamic, concurrent, and cache-efficient hash table for streaming k-mer operations. *openRxiv*, February 2026. doi:10.64898/2026.02.13.705625.

23 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.

24 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.

25 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.

26 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.

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

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

29 Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Carl Kingsford. Compact universal k-mer hitting sets. In *Algorithms in Bioinformatics*, page 257–268. Springer International Publishing, 2016. doi:10.1007/978-3-319-43681-4_21.

30 Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Carl Kingsford. Designing small universal k-mer hitting sets for improved analysis of high-throughput sequencing. *PLOS Computational Biology*, 13(10):e1005777, October 2017. doi:10.1371/journal.pcbi.1005777.

31 David Pellow, Lianrong Pu, Bariş Ekim, Lior Kotlar, Bonnie Berger, Ron Shamir, and Yaron Orenstein. Efficient minimizer orders for large values of k using minimum decycling sets. *Genome Research*, August 2023. doi:10.1101/gr.277644.123.

32 Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38:i185–i194, June 2022. doi:10.1093/bioinformatics/btac245.

33 Giulio Ermanno Pibiri and Rob Patro. Optimizing sparse and skew hashing: faster k-mer dictionaries. *Bioinformatics*, 42, July 2026. doi:10.1093/bioinformatics/btag264.

34 Arang Rhie, Sergey Nurk, Monika Cechova, Savannah J. Hoyt, Dylan J. Taylor, Nicolas Altemose, Paul W. Hook, Sergey Koren, Mikko Rautiainen, Ivan A. Alexandrov, Jamie Allen, Mobin Asri, Andrey V. Bzikadze, Nae-Chyun Chen, Chen-Shan Chin, Mark Diekhans, Paul Flicek, Giulio Formenti, Arkarachai Fungtammasan, Carlos Garcia Giron, Erik Garrison, Ariel Gershman, Jennifer L. Gerton, Patrick G. S. Grady, Andrea Guarracino, Leanne Haggerty, Reza Halabian, Nancy F. Hansen, Robert Harris, Gabrielle A. Hartley, William T. Harvey, Marina Haukness, Jakob Heinz, Thibaut Hourlier, Robert M. Hubley, Sarah E. Hunt, Stephen Hwang, Miten Jain, Rupesh K. Kesharwani, Alexandra P. Lewis, Heng Li, Glennis A. Logsdon, Julian K. Lucas, Wojciech Makalowski, Christopher Markovic, Fergal J. Martin, Ann M. Mc Cartney, Rajiv C. McCoy, Jennifer McDaniel, Brandy M. McNulty, Paul Medvedev, Alla Mikheenko, Katherine M. Munson, Terence D. Murphy, Hugh E. Olsen, Nathan D. Olson, Luis F. Paulin, David Porubsky, Tamara Potapova, Fedor Ryabov, Steven L. Salzberg, Michael E. G. Sauria, Fritz J. Sedlazeck, Kishwar Shafin, Valery A. Shepelev, Alaina Shumate, Jessica M. Storer, Likhitha Surapaneni, Angela M. Taravella Oill, Françoise Thibaud-Nissen, Winston Timp, Marta Tomaszkiewicz, Mitchell R. Vollger, Brian P. Walenz, Allison C. Watwood, Matthias H. Weissensteiner, Aaron M. Wenger, Melissa A. Wilson, Samantha Zarate, Yiming Zhu, Justin M. Zook, Evan E. Eichler, Rachel J. O'Neill, Michael C. Schatz, Karen H. Miga, Kateryna D. Makova, and Adam M. Phillippy. The complete sequence of a human Y chromosome. *Nature*, 621(7978):344–354, August 2023. doi:10.1038/s41586-023-06457-y.

35 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.

36 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.

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

38 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.

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

40 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.

41 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.