Journal Title Here, 2022, pp. 1–10

doi: DOI HERE
Advance Access Publication Date: Day Month Year
Paper

PAPER

GreedyMini: Generating low-density DNA minimizers

Shay Golan,[1,2†] Ido Tziony,[3†] Matan Kraus,[3] Yaron Orenstein[3,4*] and Arseny Shur[3*]

[1]Department of Computer Science, University of Haifa, Haifa, 3498838, Israel, [2]Efi Arzi School of Computer Science, Reichman University, Herzliya, 4610101, Israel, [3]Department of Computer Science, Bar-Ilan University, Ramat Gan, 5290002, Israel and [4]The Mina and Everard Goodman Faculty of Life Sciences, Bar-Ilan University, Ramat Gan, 5290002, Country
*Corresponding author. shur@datalab.cs.biu.ac.il, yaron.orenstein@biu.ac.il. †These authors contributed equally to this work.
FOR PUBLISHER ONLY Received on Date Month Year; revised on Date Month Year; accepted on Date Month Year

Abstract

Minimizers is the most popular k-mer selection scheme in algorithms and data structures analyzing high-throughput sequencing (HTS) data. In a minimizers scheme, the smallest k-mer by some predefined order is selected as the representative of a sequence window containing w consecutive k-mers, which results in overlapping windows often selecting the same k-mer. Minimizers that achieve the lowest frequency of selected k-mers over a random DNA sequence, termed the expected density, are desired for improved performance of HTS analyses. Yet, no method to date exists to generate minimizers that achieve minimum expected density. Moreover, for k and w values used by common HTS algorithms and data structures there is a gap between the densities achieved by existing selection schemes and a recent theoretical lower bound. Here, we present GreedyMini, a toolkit of methods to generate minimizers with low expected or particular density, to improve minimizers, to extend minimizers to larger alphabets, k, and w, and to measure the expected density of a given minimizer efficiently. We demonstrate over various combinations of k and w values, including those of popular HTS methods, that GreedyMini can generate DNA minimizers that achieve expected densities very close to the lower bound, and both expected and particular densities much lower compared to existing selection schemes. Additionally, we show that the k-mer rank-retrieval time by GreedyMini is comparable to that of common k-mer hash functions. We expect GreedyMini to improve the performance of many HTS algorithms and data structures and advance the research of k-mer selection schemes.

Key words: Minimizers, Forward local selection schemes, High-throughput sequencing, Universal hitting sets, k-mers.

1. Introduction

Minimizers are used in many bioinformatic applications, including sequence alignment, genome assembly, and data compression [1, 2, 3]. Minimizers are schemes that select representative sets of k-mers from sequences so that among every w consecutive k-mers the smallest k-mer according to some predefined order is selected. By efficiently identifying and indexing these k-mers, the computational complexity and memory requirements associated with processing large genomic datasets can be significantly reduced.

Minimizers are commonly evaluated by their expected or particular density. The expected density is the expected frequency of selected k-mers within a random DNA sequence, while the particular density is the frequency of selected k-mers in a specific DNA sequence [2]. Low density indicates a sparse and efficient selection, which enhances the performance of high-throughput sequencing (HTS) algorithms and data structures by reducing their runtime and memory usage. Thus, minimizers with low density are desired.

Several methods were developed to generate orders leading to low-density minimizers. Traditional lexicographical order often results in high density [4, 5]. Universal hitting sets (UHSs) were introduced to design orders for low-density minimizers. But, expensive heuristics are required to generate UHS-based orders limiting them to $k \leq 15$ [6, 7, 8]. Decycling-based minimizers can be applied to any k value and achieve the lowest expected density compared to existing selection schemes to date [9]. Frequency-based orders can be used as a simple technique to generate minimizers with low particular density [10]. More recently, DeepMinimizer was developed to utilize machine learning to distribute selected k-mers more evenly, directly targeting low particular density [11]. To date, no method solves the problem of generating a minimum-expected-density minimizers scheme, and there is a gap between the theoretical lower bound and the densities achieved by existing selection schemes [12]. Some minimizers, such as Miniception [13] and syncmers [14], and other k-mer selection schemes, such as mod-minimizer [15] and strobemers [16], employ "internal" minimizers and can benefit from the use of low-density minimizers.

Here, we present GreedyMini, a toolkit of novel methods to generate minimizers with low expected or particular density,

© The Author 2022. Published by Oxford University Press. All rights reserved. For permissions, please e-mail: journals.permissions@oup.com

to locally improve a given minimizer, to extend minimizers to larger alphabets, k, and w, and to measure the expected density of a given minimizer efficiently. We prove density guarantees for GreedyMini's extensions to larger alphabets, and larger k. We combine these innovations into several pipelines to generate low-density DNA minimizers. We demonstrate over various combinations of k and w values, which are most practical for HTS analyses, that the pipelines generate minimizers that achieve lower density compared to existing selection schemes and very close to a recent theoretical lower bound [12] (e.g., within 1% for $w = k - 1$) and at comparable k-mer rank-retrieval time as common k-mer hash functions. We expect GreedyMini to improve the runtime and memory usage of HTS algorithms and data structures and to lead to further advancements in generating minimizers and k-mer selection schemes with low density.

2. Preliminaries

For integers $i \geq 0$ and $j \geq i$ we denote $[i, j) = \{i, \ldots, j-1\}$. Let $\Sigma = [0, \sigma)$ be an alphabet and $S \in \Sigma^+$ be a non-empty string over Σ. $S[i]$ denotes the letter at position i (the leftmost letter is $S[0]$), and $S[i, j)$ refers to the substring covering the interval $[i, j)$ of positions, so that $S = S[0, |S|)$. Substrings of the form $S[0, j)$ and $S[i, |S|)$ are *prefixes* and *suffixes* of S, respectively. *Selection schemes* (defined below) operate by selecting substrings of small length k, called *k-mers*, in longer substrings called *windows*. We write *n-window* (and n-prefix, n-suffix, and n-string) to indicate length. Our algorithms assume the common unit-cost word-RAM model. If any integer function $g(\sigma, k, w)$ appears as the time or space complexity of an algorithm, its values are assumed to fit into $O(1)$ machine words; σ is treated as a constant. For a boolean expression B, $[B]$ equals 1 if B is true and 0 otherwise.

A *local (selection) scheme* with parameters (σ, k, w) is a map $f : \Sigma^{w+k-1} \to [0, w)$. This map acts on strings over Σ, selecting one position in each $(w+k-1)$-window so that in a window $v = S[i, i + w + k - 1)$ the position $i + f(v)$ in S is selected.

The most popular metric to evaluate a local scheme by is the density of selected positions. Let $f(S)$ denote the set of positions selected in a string S by a local scheme f. The *particular density of f on S* is $d_f(S) = \frac{|f(S)|}{|S|-k+1}$. The *expected density of f* is the limit $d_f = \lim_{n\to\infty} \frac{1}{\sigma^n} \sum_{S\in\Sigma^n} d_f(S)$. The *density factor* of f is a normalization of d_f by a factor of $(w+1)$, i.e., $df_f = (w + 1)d_f$.

Let π be a permutation (= a linear order) of the set of all σ-ary k-mers. We view π as a bijection $\pi : [0, \sigma^k) \to \Sigma^k$ and say that a k-mer u has *rank* i, denoted by $\mathsf{rank}_\pi(u) = i$, if $\pi(i) = u$. The *minimizer* (π, w) is a local scheme that maps each $(w+k-1)$-window to the starting position of its minimum-rank k-mer with ties broken to the left.

A subset $H \subseteq \Sigma^k$ is a *universal hitting set* (UHS) for w if every $(w+k-1)$-window contains at least one k-mer from H. Let π be a linear order on Σ^k and let i be such that $H = \{u \mid \mathsf{rank}_\pi(u) < i\}$ is a UHS for w. Every permutation π' such that $\pi'(j) = \pi(j)$ for all $j \in [0, i)$ generates the same minimizer: $(\pi', w) = (\pi, w)$. Accordingly, we define a *UHS order on* Σ^k (for w) to be any bijection $\rho : [0, |H|) \to H$, where $H \subseteq \Sigma^k$ is a UHS for w. Any completion of ρ to a permutation π of Σ^k defines the same minimizer, which we denote by (ρ, w). We define $\mathsf{rank}_\rho(u)$ as above if $u \in H$ and set $\mathsf{rank}_\rho(u) = \infty$ otherwise. Since Σ^k is a UHS for any w, a permutation of Σ^k is a special case of a UHS order.

For a given minimizer $f = (\rho, w)$, a $(w+k)$-window v is *charged* if its minimum-rank k-mer is either its prefix or its *unique suffix* (i.e., the k-suffix of v having no other occurrence in v); otherwise, v is *free*. An important observation is that every string S contains exactly $|f(S)| - 1$ (not necessarily distinct) charged $(w+k)$-windows (Lemma 6 in [2]). Since all possible n-strings have, in total, the same number of occurrences of each $(w+k)$-window, the expected density d_f of a minimizer equals the fraction of charged windows in Σ^{w+k} [5, 13].

3. Methods

3.1. Generating minimizer orders greedily

We present a novel randomized algorithm GreedyE (Algorithm 1, E stands for expected) to generate low-density minimizers. Given parameters (σ, k, w), GreedyE generates a UHS order ρ on Σ^k for w, assigning ranks to k-mers one by one, starting from rank 0 and stopping when the k-mers with assigned ranks constitute a UHS. GreedyE maintains a score function for all unranked k-mers. At each iteration, GreedyE creates a pool of low-scored k-mers, chooses a random k-mer from it, assigns the next rank to the chosen k-mer, and updates the scores of all unranked k-mers.

During an iteration, a $(w+k)$-window is *live* if all its k-mers are unranked. For an unranked k-mer u, let X_u be the set of all live windows containing u, and let $Y_u \subseteq X_u$ consist of live windows containing u as a prefix or as a unique suffix. Then, $\mathsf{score}(u) = |Y_u|/|X_u|$. The intuition behind selecting a low-scored k-mer is the following: choosing u to have the next rank clarifies the status (free or charged) of the windows from X_u and only of them, and those from Y_u are charged. Thus, $\mathsf{score}(u)$ locally approximates the fraction of charged windows among all windows, which is the fraction we want to minimize.

Algorithm 1 : GreedyE: Generating a low-density UHS order

Require: alphabet size σ, k-mer length k, window size w
Ensure: UHS order ρ for w on Σ^k

```
rank ← [∞, ..., ∞]                          ▷ initialize ranks of all k-mers
X, Y ← init_counts(w, k, σ)                 ▷ X[u] = |X_u|, Y[u] = |Y_u|
                                                  for each k-mer u
for r ← 0 to σ^k − 1 do                     ▷ current rank to be assigned
    u ← select_candidate(X, Y)              ▷ select k-mer u among
                                                the k-mers with (almost) lowest |Y_u|/|X_u|
    if u = null then                        ▷ no live windows, X = [0, ..., 0]
        return rank                         ▷ ρ is defined by the ranks of k-mers
    end if
    X, Y ← update_counts(X, Y, u, rank)     ▷ for next iteration
    rank[u] ← r                             ▷ assign rank r to the selected k-mer u
end for
```

In our implementation, we define low-scored k-mers as follows. For each run of GreedyE we sample a parameter α from a user-defined range: $\alpha \sim \mathrm{LogUniform}(\alpha_{min}, \alpha_{max})$. At each iteration of GreedyE, an unranked k-mer u is considered low-score if its *inverted score* $\frac{\mathsf{X}[u]-\mathsf{Y}[u]}{\mathsf{Y}[u]}$ is at least αI, where I is the maximum inverted score at this iteration. With $\alpha_{min} = \alpha_{max} = 1$, GreedyE selects a lowest-score k-mer at each iteration but still uses randomization when several k-mers are tied for

the lowest score. Decreasing α_{min} and α_{max} expands the search space of low-density minimizers at no additional space or runtime costs, and thus has the potential to improve the best result over multiple runs.

The natural computational requirements for implementing GreedyE are: $\Omega(\sigma^k)$ space to store the order; $\Omega(\sigma^k)$ time per iteration to select a low-score k-mer; $\Omega(w\sigma^{k+w})$ time to scan each window once. Theorem 1 presents our implementation, which meets all these lower bounds up to additional $O(w)$ space. We prove the theorem and give the pseudocode of non-trivial auxiliary functions in Supplementary Section S4.

Theorem 1 GreedyE *can be implemented to run in $O(\sigma^k + w)$ space and $O(|H|(\sigma^k + w\sigma^w))$ time, where H is the constructed UHS.*

Remark 1 *(i) The size of every UHS is between $\frac{\sigma^k}{k}$ and σ^k. (ii)* GreedyE *can be implemented with the running time $O(|H|w\sigma^{2k})$, which is linear in w but has heavier dependence on k. We found no practical use for this implementation, so we only briefly describe it in Supplementary Section S4.*

To generate minimizers of low *particular* density with respect to a specific input string S, we present an analog of GreedyE, called GreedyP. In this case, each window has a weight equal to the number of its occurrences in S. GreedyP computes and stores the windows occurring in S with their weights at an additional expense of $O(|S|)$ time and $O(\min\{|S|, \sigma^{w+k}\})$ space, since the number of $(w+k)$-windows of positive weight is bounded by the observed windows in S and by σ^{w+k}. Then, the only difference with GreedyE (Algorithm 1) is that the arrays X and Y contain *sums of weights* of windows rather than the numbers of windows. For the case where $|S| \ll \sigma^{w+k}$, GreedyP can be implemented more efficiently based on a linked list of windows with positive weight.

3.2. Improving minimizers by hill climbing

We apply a randomized version of the standard local search technique called *hill climbing* to improve a given minimizer (ρ, w), where ρ is a UHS order for w. At each iteration, the search algorithm samples a random rank r such that both r and $r+1$ are assigned in ρ and calls the function Swap(ρ, w, r), defined as follows. Swap(ρ, w, r) generates the UHS order ρ' from ρ by swapping the k-mers with ranks r and $r+1$ and calculates $\Delta_{\rho,\rho'} \triangleq d_{(\rho,w)} - d_{(\rho',w)}$. If $\Delta_{\rho,\rho'} > 0$, Swap replaces ρ with ρ'; if $\Delta_{\rho,\rho'} < 0$, Swap does nothing; if $\Delta_{\rho,\rho'} = 0$, Swap replaces ρ with ρ' with probability 0.5. The search stops when a user-defined time limit or number of iterations is reached.

Theorem 2 presents the complexity of our two efficient implementations of Swap. Our default implementation (a) uses recursive depth-first traversal (SwapDFS), while (b) is designed for the case of large w and based on dynamic programming (SwapDP). The proof of Theorem 2 and the pseudocodes of SwapDFS and SwapDP are presented in Supplementary Section S3.

Theorem 2 *Let ρ be a UHS order on Σ^k for w, and let the ranks $r, r+1$ be assigned in ρ. The function* Swap(ρ, w, r) *can be implemented in either (a) $O(\sigma^k + \sigma^w)$ time and $O(w)$ additional space or (b) $O(w\sigma^k)$ time and $O(\sigma^k)$ additional space.*

3.3. Extending orders to larger σ, k, and w

A local scheme f with the parameters (σ, w, k) can be extended to a local scheme f' with the parameters (σ, w, k'), where $k' > k$, by a simple rule [17]: in every $(w+k'-1)$-window, f' picks the position chosen by f in the $(w+k-1)$-prefix of this window. Trivially, $d_f = d_{f'}$. In a similar way, a local scheme f_γ with the parameters $(\sigma\gamma, w, k)$ can be built from f. Namely, the alphabet is partitioned into σ classes, and in every window v f_γ picks the position chosen by f in the window $\bar{v}$, obtained from v by replacing each letter with its class. Here as well, $d_f = d_{f_\gamma}$.

Such extensions may suffice for practice. But, their main theoretical limitation is that f' and f_γ do not inherit the property "to be a minimizer" from f. Namely, if f is a minimizer, then f_γ is never a minimizer, while the status of f' depends on f and w. In particular, if $w > k'$, then f' is not a minimizer for any f. See Supplementary Section S5 for a detailed discussion.

We upgrade these trivial extensions to obtain minimizer-to-minimizer extensions, and prove their density guarantees. We define *dual minimizers* as local schemes that act like minimizers but break ties to the right. For a minimizer f, we denote the dual minimizer using the same order as f by $f^{\leftarrow}$. Notice that d_f and $d_{f^{\leftarrow}}$ are close: only the windows containing their minimum-rank k-mer as a non-unique prefix or a non-unique suffix are charged differently (e.g., the average difference in expected density was less than 0.15% over 1000 random minimizers for $\sigma = 2$, $k = 8$, and $w = 12$). Given a UHS order ρ for w on Σ^k, we define its *mirror* as the UHS order $\rho^{\leftarrow}$ such that $\mathsf{rank}_{\rho^{\leftarrow}}(u) = \mathsf{rank}_\rho(\overleftarrow{u})$ for all $u \in \Sigma^k$, where $\overleftarrow{u}$ is u reversed (Figure 1). Lemma 1 connects mirrors to dual minimizers.

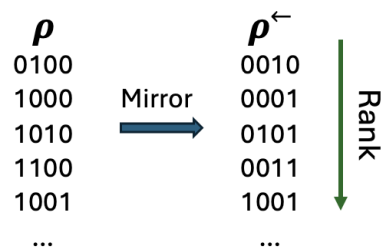

Fig. 1. An example of a binary order ρ on 4-mers and its mirror $\rho^{\leftarrow}$.

Lemma 1 If ρ is a UHS order for w, then $d_{(\rho^{\leftarrow},w)} = d_{(\rho,w)^{\leftarrow}}$.

Proof If a $(w+k)$-window v is charged by the scheme $(\rho, w)^{\leftarrow}$, its k-mer u with the minimum ρ-rank is either a suffix or a unique prefix of v. Hence, in the window $\overleftarrow{v}$ the k-mer $\overleftarrow{u}$ has the minimum $(\rho^{\leftarrow})$-rank and is either a prefix or a unique suffix; respectively, $\overleftarrow{v}$ is charged by $(\rho^{\leftarrow}, w)$. As all steps are reversible, we get a bijection between the sets of charged windows of the two schemes. □

We transform an arbitrary minimizer f with the parameters (σ, k, w) to a minimizer f' with the parameters $(\sigma\gamma, k, w)$ for an integer $\gamma > 1$ as follows. Let $\Gamma = [0, \gamma)$ and consider the alphabet $\Sigma \times \Gamma = \{\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right] \mid a \in \Sigma, b \in \Gamma\}$ of size $\sigma\gamma$. Every string $S = \left[\begin{smallmatrix} a_0 \\ b_0 \end{smallmatrix}\right]\left[\begin{smallmatrix} a_1 \\ b_1 \end{smallmatrix}\right] \cdots \left[\begin{smallmatrix} a_{n-1} \\ b_{n-1} \end{smallmatrix}\right] \in (\Sigma \times \Gamma)^n$ has two *projections* $S_\Sigma = a_0 \cdots a_{n-1} \in \Sigma^n$ and $S_\Gamma = b_0 \cdots b_{n-1} \in \Gamma^n$. Given a UHS order ρ for w on Σ^k and a linear order τ on Γ^k, we define a UHS order $\rho \times \tau$ for w on $(\Sigma \times \Gamma)^k$ by setting $\mathsf{rank}_{\rho\times\tau}(u) =$

$\gamma^k \mathsf{rank}_\rho(u_\Sigma) + \mathsf{rank}_\tau(u_\Gamma)$; in particular, $\mathsf{rank}_{\rho\times\tau}(u) = \infty$ iff $\mathsf{rank}_\rho(u_\Sigma) = \infty$. The following theorem presents a minimum density guarantee over the minimizers $(\rho\times\tau, w)$ and $(\rho^{\leftarrow}\times\tau, w)$. Its proof is provided in Supplementary Section S6.

Theorem 3 *Let ρ be a UHS order on Σ^k for w and τ be a linear order on Γ^k. Then,* $\min\{d_{(\rho\times\tau,w)}, d_{(\rho^{\leftarrow}\times\tau,w)}\} \leq (d_{(\rho,w)} + d_{(\rho,w)^{\leftarrow}})/2$.

We use a similar approach to obtain the same density guarantee for the extension of a minimizer with parameters (σ, k, w) to a minimizer with the parameters $(\sigma, k{+}1, w)$. Given a UHS order ρ on Σ^k for w, we define its *extensions* ρ_1 and ρ_2 on Σ^{k+1} as follows. For $u \in \Sigma^k$ and $a \in \Sigma$, we set $\mathsf{rank}_{\rho_1}(ua) = \sigma{\cdot}\mathsf{rank}_\rho(u) + a$ and $\mathsf{rank}_{\rho_2}(au) = \sigma{\cdot}\mathsf{rank}_{\rho^{\leftarrow}}(u) + \sigma - a - 1$. We prove the next theorem in Supplementary Section S7.

Theorem 4 *Let ρ be a UHS order on Σ^k for w, and let ρ_1 and ρ_2 be its extensions. Then,* $\min(d_{(\rho_1,w)}, d_{(\rho_2,w)}) \leq (d_{(\rho,w)} + d_{(\rho,w)^{\leftarrow}})/2$.

3.4. Measuring the density of a given minimizer

A standard approach to measure the expected density of a given selection scheme is to generate a long (pseudo)random string and count sampled positions. However, the exact calculation of the expected density is preferred whenever it is feasible. As was pointed out in [5], the exact density can computed by processing a de Bruijn sequence of order $w+k$ instead of a random string, as such a de Bruijn sequence contains each $(w+k)$-window exactly once. In this way, the expected density can be computed in $O(\sigma^{w+k})$ time and $O(w+k)$ space using a fast shift rule for de Bruijn sequences [18, 19].

We present two new algorithms, DenDFS and DenDP, that calculate the exact expected density of a given minimizer more efficiently. Their complexity is given by Theorem 5 (a) and (b), respectively. The algorithms and the proof of Theorem 5 are presented in Supplementary Section S2.

Theorem 5 *Let ρ is a UHS order on Σ^k for w and let H be its UHS. The expected density of the minimizer (ρ, w) can be computed in either (a) $O(|H|\sigma^w)$ time and $O(w)$ space, or (b) $O(w|H|\sigma^k)$ time and $O(\sigma^k)$ space.*

Remark 2 *(i) Since the size of H is at most σ^k and can be as small as $\frac{\sigma^k}{k}$,* DenDFS *is faster than the algorithms that process a de Bruijn sequence. In our benchmarking (Section 4.1), more than 90% of the binary UHS orders satisfy $|H| \leq 4 \cdot \frac{2^k}{k}$. (ii)* DenDP *avoids exponential dependence on w and thus computes the density for very large windows at a modest space expense.*

3.5. GreedyMini toolkit and pipelines

We combined the novel techniques from Sections 3.1–3.4 into a toolkit called GreedyMini. Its primary purpose is the generation of low-density DNA minimizers for given parameters k and w.

The main tools are GreedyE and GreedyP (Section 3.1), implemented for the *binary* alphabet. GreedyE takes the parameters k, w, a range $(\alpha_{min}, \alpha_{max})$, and returns a UHS order ρ on $\{0,1\}^k$ for w such that the minimizer (ρ, w) has low expected density. GreedyP has the same input parameters plus a sequence S, and outputs a UHS order ρ on $\{0,1\}^k$ such that the minimizer (ρ, w) has low particular density for S.

The tools SwapDFS and SwapDP (Section 3.2) take the parameter w and a UHS order ρ on $\{0,1\}^k$ for w as input and return an order ρ' on $\{0,1\}^k$ such that $d_{(\rho',w)} < d_{(\rho,w)}$, unless no such order ρ' was found, and then ρ is returned. By default, SwapDFS is used, while SwapDP is intended to cover large values of w.

The tools DenDFS and DenDP (Section 3.4) compute the exact expected density of *binary* minimizers with DenDFS being the default method and DenDP covering large values of w.

Finally, the toolkit includes auxiliary functions described in Section 3.3. Mirror receives a binary UHS order ρ and returns the order $\rho^{\leftarrow}$; Extend-k receives a binary minimizer and extends it to a larger value of k; Extend-σ receives a binary minimizer and extends it to the DNA alphabet (taking $\sigma = \gamma = 2$ in the extension definition, with τ being the lexicographic order).

We implemented several pipelines as part of GreedyMini.

GM-expected: *Generating a DNA minimizer for (k, w)*

To construct a DNA minimizer with low *expected* density for given k and w, we proceed in three stages (Supplementary Figure S2A).

Generate: Given k and w, run multiple instances (by default 4096) of GreedyE$(2, k, w)$; the default range $(\alpha_{min}, \alpha_{max})$ to sample the parameter α is $(0.94, 0.9996)$. Report a lowest-density minimizer or its mirror, whichever has lower density.

Improve: For the minimizer reported by stage Generate, run an instance of SwapDFS on each available CPU core (64 on our machine) with the time limit set to the runtime of stage Generate (in our experiments, this resulted in $>400 \cdot 2^k$ Swap calls per core). Report a lowest-density minimizer.

Extend: Given the minimizer (ρ, w) from stage Improve, report the DNA minimizer Extend-$\sigma(\rho, w)$ or Extend-$\sigma(\rho^{\leftarrow}, w)$, whichever achieves lower density.

GM-particular: *Generating a DNA minimizer for (k, w) and sequence S*

To construct a DNA minimizer with low *particular* density for given k, w, and a sequence S, we perform stage Generate, replacing GreedyE by GreedyP. Then, we skip stage Improve and run stage Extend to obtain the result (Supplementary Figure S2B).

GM-improve: *Generating a DNA minimizer for large w*

If running GM-expected on the pair (k, w) is expected to take too much time due to large w, we start with a binary minimizer obtained for the pair (k, w') for some $w' < w$. We run an instance of SwapDP on each available CPU core within a manually set time limit (10 hours by default). Finally, we report the lowest-density minimizer over these runs. For the reported minimizer, we run stage Extend to obtain the final result (Supplementary Figure S2C). Additionally, GM-improve can be used to improve any binary minimizer via the use of SwapDP.

GM-k: *Generating a DNA minimizer with increased k*

If running GM-expected on the pair (k, w) resulted in a minimizer having higher density than the minimizer obtained for the pair $(k-1, w)$, we start with the binary minimizer obtained for $(k-1, w)$, apply Extend-k on it, and run stage Improve within a manually set time limit. Then, we run stage Extend to obtain the result. This pipeline is useful for the case where k is large enough and w is small enough, as running Swap takes just $O(2^w)$ time, while the runtime of an instance of GreedyE is $O(2^{2k})$.

Trivial extensions

If running SwapDP and SwapDFS for the pair (k, w) takes too much time or is even infeasible, we use trivial extensions. For a trivial k-extension, we take a binary minimizer for some parameters $k' < k$ and w, apply Extend-k, and run stage EXTEND on it. For a trivial w-extension, we take a binary order ρ obtained for some parameters k and $w' < w$, consider the minimizer (ρ, w), and run stage EXTEND on it.

4. Results

4.1. Benchmarking k-mer selection schemes

We used GreedyMini to generate low-density DNA minimizers for various k, w combinations and benchmarked the results against the minimizers constructed by the state-of-the-art methods: double decycling [9], miniception [13], and the recent open-closed syncmer [20], which mixes the ideas of the previous two. For these methods we used the implementations from the mod-minimizer GitHub repository [21] and measured their density using a random 4-ary 10^7-string. We excluded from the comparison decycling [9], DOCKS [7], and PASHA [8] methods as they are clearly outperformed by double decycling [9].

To benchmark in the mode "fixed w, variable k", we took $w = 8$ and $w = 12$. For $w = 8$, we ran GM-expected for all $k \in [3, 18)$ and made a trivial k-extension up to $k = 30$. For $w = 12$, we ran GM-expected for all $k \in [3, 17)$ and extended trivially up to $k = 26$. Similarly, to benchmark in the mode "fixed k, variable w", we took $k = 8$ and $k = 12$. For $k = 8$, we ran GM-expected for all $\in [3, 20)$, and made a trivial w-extension up to $w = 100$. For $k = 12$, we ran GM-expected for all $\in [3, 17)$, and then ran GM-improve subsequently for $w = 20, 25, 30$, taking the previous result as the starting point. We then trivially extended the resulting minimizer up to $w = 100$.

In addition, we ran GM-expected over all (k, w) pairs with $3 \leq k, w \leq 15$ and focused on the mode "$w \approx k$". There were a few cases where the density obtained by GM-expected for (k, w) was higher than the one obtained for $(k - 1, w)$. In each such case, we ran GM-k to improve the result.

Finally, we addressed specific (k, w) pairs used in popular HTS methods: $(9, 16)$ (KMC 3) and $(15, 17)$ (Kraken) were processed by GM-expected; for $(7, 22)$ and $(7, 49)$ (both KMC 2) we made a trivial w-extension from $w = 15$; for $(21, 11)$ (SSHash), $(29, 11)$ (Giraffe), and $(31, 5)$ (Kraken 2) we made a trivial k-extension from $k = 15$; for $(19, 30)$ (GraphAligner) we made a trivial k-extension from $k = 12$.

The expected density of the resulting DNA minimizers was computed as follows: (i) $k + w \leq 26$, except for trivial extensions: exact computation by DenDFS/DenDP; (ii) $27 \leq k + w \leq 30$, except for trivial extensions: the binary density is computed exactly, and the number of bad/good $(k + w)$-windows (Supplementary Section S6) is estimated from a sample of 10^8 windows over Γ; (iii) $k + w \geq 31$, plus all trivial extensions: the binary density is computed exactly, then the upper bound of Theorem 3 is reported.

We also ran GM-particular to generate minimizers with low particular density for a DNA sequence of 10^6 nucleotides from Chromosome X for $w = 12$ and all $k \in [3, 17)$, and for $k = 8$ and all $w \in [3, 20)$. We compared the results to the same set of benchmarked selection schemes; our attempts to compare to DeepMinimizer [11] and polar sets [22] failed, because their GitHub implementations raised errors and lacked documentation (Supplementary Section S9). We report the minimizer construction runtime and maximum memory usage of GM-expected and GM-particular in Supplementary Section S10.

4.2. Expected and particular density results

For fixed w (Figures 2A,B), GM-expected outperformed all other minimizers, losing to open-closed syncmer in a single point $(12, 26)$. For $k = 8$ (Figure 2C), GM-expected outperforms all other minimizers by a large margin, even though it uses the trivial w-extension.. For $k = 12$ (Figure 2D), trivial w-extension of the GM-expected from $w = 16$ is not enough: its result is nearly tied with the result of double decycling for $w > 20$. However, the use of GM-improve results in a clear gap of $2.40\% - 3.86\%$. We observed similar trends for particular density (Figures 2E,F), where the best performers were GM-particular and GM-expected, with GM-particular achieving clearer superiority for $w + k > 24$.

In the case $w \in \{k, k - 1\}$ (Figures 2G,H), GM-expected generated minimizers that have expected density within 1% of the theoretical lower bound for forward schemes [12], which constitute a wide class of local schemes including all minimizers.

4.3. Density comparisons for popular HTS methods

To emphasize the relevance and potential impact of GreedyMini, we calculated the expected density factors of minimizers generated by GM-expected over (k, w) combinations used by popular HTS methods [23, 24, 25, 26, 27, 28, 29, 30, 31, 32]. We compared the results of GreedyMini to those of double decycling, miniception, open-closed syncmer, and random minimizer schemes.

For the combinations with $w > k$ in four out of five cases, GreedyMini achieved the best result (Figure 3A, Supplementary Table S1). It loses to double decycling and open-closed syncmer schemes only for the pair $(19, 30)$ used by GraphAligner. Among four considered combinations with $k > w$, GreedyMini shows the best result for three and second best (after open-closed syncmer) for the remaining one (Supplementary Table S1).

However, among all known forward schemes, the best results in the case $k > w$ are achieved by the mod-minimizer scheme [15]. This scheme is not a minimizer, but uses a minimizer (for some $k' < k$) as an intermediate step. Thus, we plugged the minimizers obtained by GreedyMini and all benchmarked methods into the mod-minimizer scheme, using the parameter $r = 4$ as in the original setting [20]. We measured the density of obtained schemes over a 10^7-long random DNA sequence. In all four cases, GreedyMini showed the best result (Figure 3B, Supplementary Table S2).

4.4. k-mer rank-retrieval times

We next evaluated the sampling time for the DNA minimizers produced by GreedyMini. Sampling a string S consists of two tasks: retrieving the ranks of all k-mers and computing the minimum rank in a sliding window. The second task is independent of the minimizer used, so we focused on the rank-retrieval time.

We ran a code that assigns ranks to k-mers on a random DNA sequence of $3 \cdot 10^9$ nucleotides, and calculated the average runtime over 10 runs for robust estimation. To simulate an order generated by GM-expected or GM-particular, we stored a binary order in memory and extended it by Extend-σ. For benchmarking, we used two ranking algorithms that do not require storing an order in memory: a trivial XOR-based hash function, which is used by some minimizers to create a

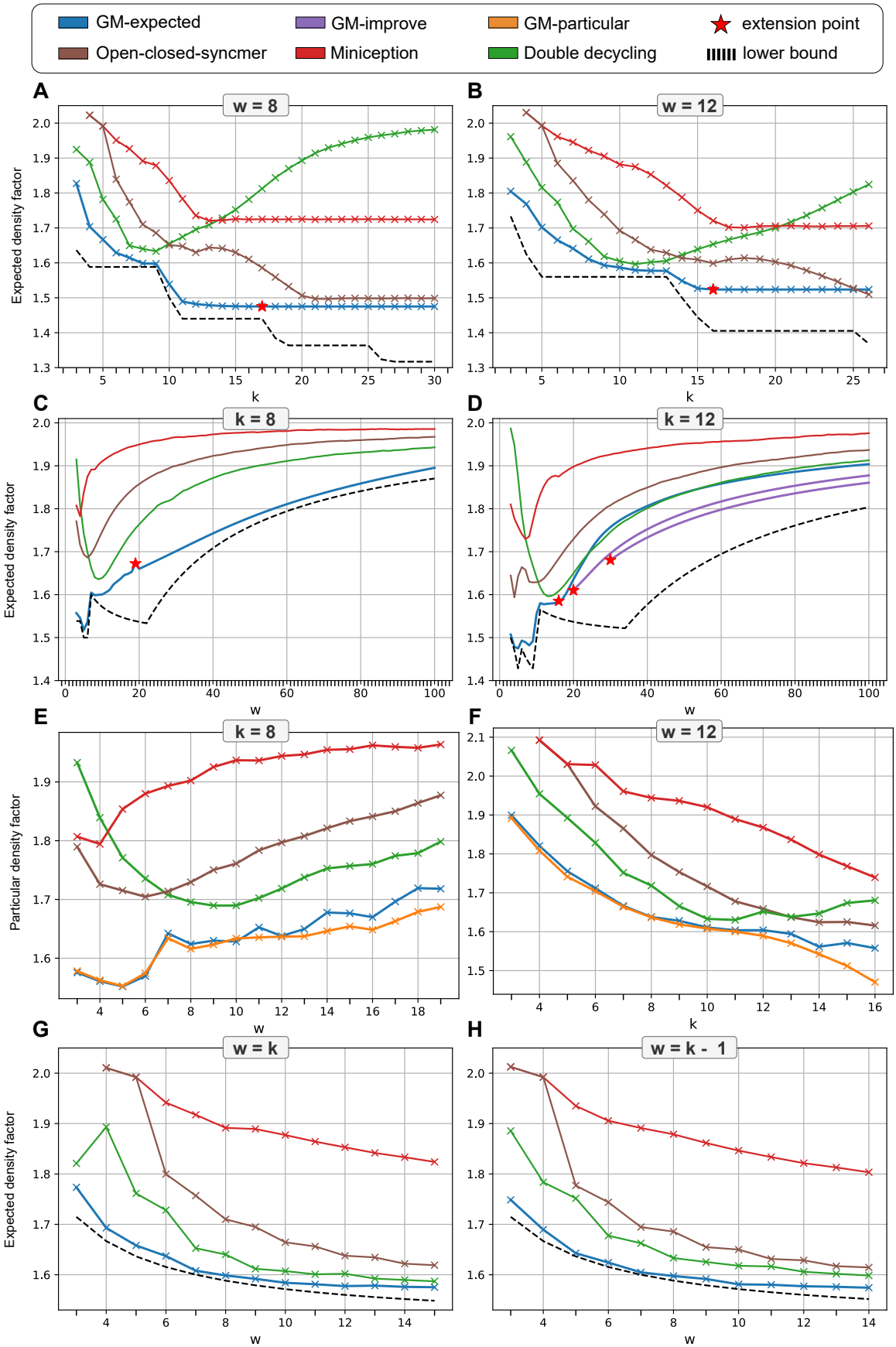

Fig. 2. Performance of various DNA k-mer selection schemes over various (k, w) values. (A-D) Expected density comparisons over fixed k or w. (E,F) Particular density comparisons over 1 million nucleotides from Chromosome X. (G,H) Expected density comparisons over $w \in \{k, k-1\}$.

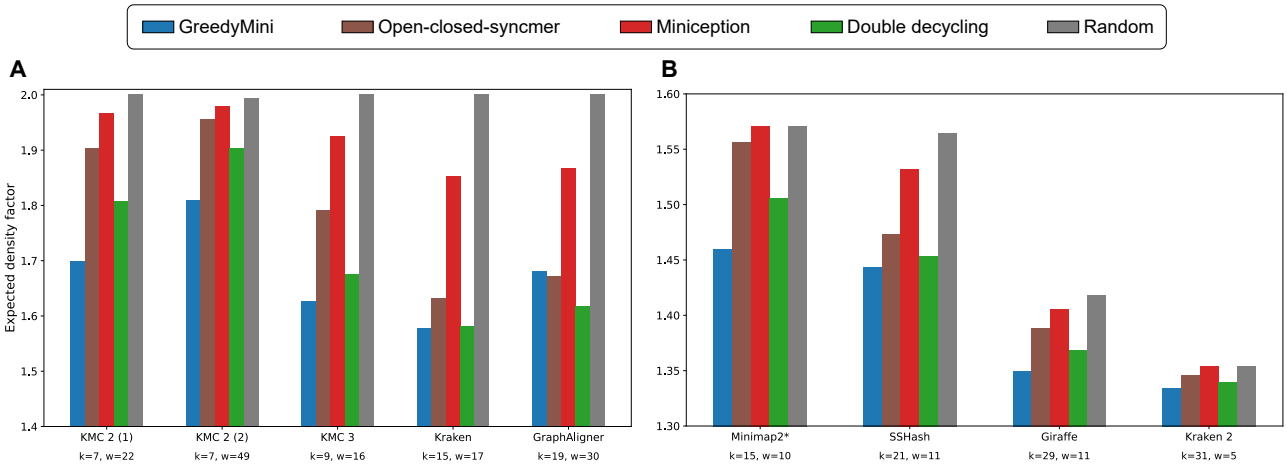

Fig. 3. Density comparison over popular HTS methods. (A) Comparisons with other minimizers for $k < w$. (B) Comparison with other minimizers when plugged into a mod-minimizer scheme for $k > w$. *These (k, w) values also apply to Minimap [28] and MetaProb 2 [30]. We calculated the densities over a 10^7-long random DNA sequences.

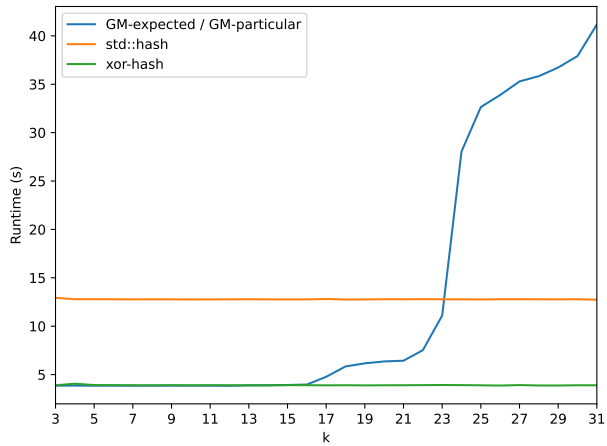

Fig. 4. Runtime to assign ranks to k-mers over a $3 \cdot 10^9$-long random DNA sequence, averaged over 10 runs.

pseudo-random order, and `std::hash`, which is the default hash implementation in C^{++}. We ran our code on an AMD Ryzen™ 7 7800x3D (L1/L2/L3 cache size: 512KB/8MB/96MB) alongside 32GB of DDR5 RAM.

For values for which GM-expected generates DNA minimizers in a reasonable runtime (i.e., $k \leq 15$), it is also very fast to retrieve k-mer ranks using the resulting minimizers (Figure 4). The rank-retrieval times for GreedyMini match those for the XOR-based hash function (less than 4 seconds over 3 billion nucleotides including the generation of the random input string). For $17 \leq k \leq 22$, there was a small increase in time, which still remains well below the results of `std::hash`. Only for $k > 22$, the rank-retrieval time steeply increases, presumably switching to storing (some of) the order in the slowest cache or even in the main memory. Since any order generated by GreedyMini for $k > 22$ is likely to use a trivial k-extension from some smaller k', then only the order of k'-mers needs to be stored. Thus, on the tested hardware the sampling time of the minimizers generated by GreedyMini is guaranteed to be competitive.

4.5. Effect of individual steps on density results

We next measured and analyzed the impact of different steps in GreedyMini on the obtained densities. We tested the effect of transforming binary minimizers to DNA minimizers by Extend-σ and the effect of the local improvements by Swap.

The difference in expected density between the binary minimizers generated by stages Generate and Improve of GM-expected and the DNA minimizers obtained with Extend-σ in stage Extend was less than 1% for all minimizers generated over $5 \leq k \leq 15$ and $3 \leq w \leq 15$ (Supplementary Table S3); moreover, for $k \geq 9$ this difference was well below 0.1% in all cases. Thus, we conclude that Extend-σ produces DNA minimizers with densities very similar to those of the original binary minimizers.

The contribution of Swap was minor for most (k, w) combinations we tested (with the orders generated by GreedyE). For most pairs (k, w) with $3 \leq w \leq 15$, $k \leq 6$, there was no reduction in expected density, possibly due to a global or local optimum. For $3 \leq w \leq 15$ and $7 \leq k \leq 15$, the average reduction in expected density was $0.26\% \pm 0.15\%$ with the reduction 0.67% for the pair $(13, 10)$ being the best improvement (Supplementary Table S4). In contrast, if the pair (k, w) is out of the range of GreedyE, the use of Swap in the GM-improve pipeline resulted in substantial improvements. For example, for the pair $(12, 30)$ Swap has reduced the expected density by 4.35%.

Overall, we conclude that the principal role in constructing low-density minimizers is played by GreedyE/GreedyP, with mostly marginal effect of other steps on density.

4.6. Towards optimal minimizers

As already mentioned in Section 4.2, the densities of minimizers generated by GreedyMini pipelines are close to the lower bound on the density of forward schemes [12]. Moreover, the *binary* minimizers by GreedyMini pipelines in the process hit this lower bound for the following (k, w) pairs: $(4, 3)$, $(7, 3)$, $(10, 3)$, $(5, 4)$, and $(6, 5)$. Apart from proving that, for some (k, w) combinations, minimizers are as good as general forward schemes, this result supports the conjecture that, within the range where GreedyMini is feasible to run, it generates minimizers with the density close to the lower bound. Based on

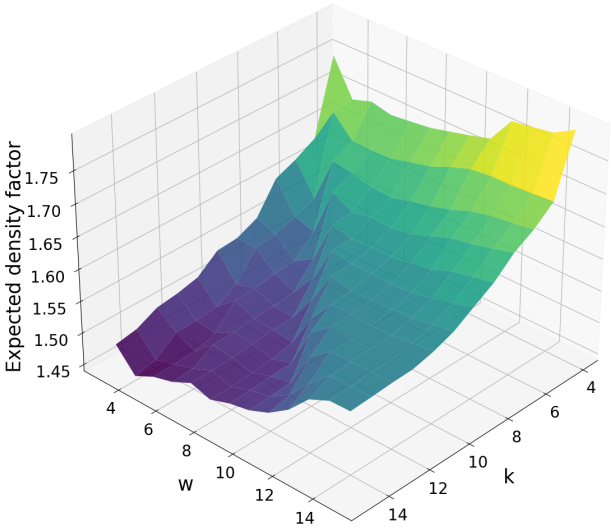

Fig. 5. Density factors of the best binary minimizers generated by GreedyMini pipelines.

this conjecture, we present in Figure 5 the landscape of density factors of our best binary minimizers as an approximation of such a landscape of minimum densities. The main features of this landscape are flat "highlands" ($w \geq k-1$), gradually ascending as w grows over k, bumpy "lowlands" ($w \leq k-3$), and a steep step between them ($k-3 \leq w \leq k-1$).

5. Discussion

In this paper, we tackled the problem of generating a minimizer that achieves low expected or particular density. We presented GreedyMini, a toolkit of novel methods to generate low-density minimizers, to improve a given minimizer, to extend to larger alphabet, k, and w, and to measure the density of a given minimizer. With these methods, we devised pipelines to generate low-density DNA minimizers for given (k, w) combinations, with the key role played by two variants of a greedy algorithm (GreedyE and GreedyP for expected and particular density, respectively).

We evaluated the performance of the minimizers generated by GreedyMini pipelines by calculating the achieved expected and particular densities, and conclude that GreedyMini pipelines generate minimizers that achieve the lowest density compared to all existing minimizer methods over a wide range of (k, w) combinations. This range includes the parameters used by popular HTS tools, such as Minimap2 [29], Kraken 2 [24], KMC 3 [26], and several others. Moreover, we observed that the achieved densities are very close to the theoretical lower bound in that range, and even hit the lower bound for forward schemes for the binary alphabet and several (k, w) pairs.

The main limitation of GreedyMini is the exponential dependence on $w+k$ of the time needed to build a minimizer by GreedyE. Thanks to our novel minimizer-to-minimizer transformation to larger alphabets (Extend-σ), this dependence is only $O(2^{2k}+w2^{w+k})$ per run of GreedyE. This is an acceptable construction time for a wide range of practical values of k and w. This range is further extended to larger k values due to Extend-k and to larger w values by Extend-w and by GM-improve. The use of Extend-σ reduces the exponential dependence of the space complexity in k to just $O(2^k)$, which is a feasible limitation. For $k = 20$, the lookup table occupies $4 \cdot 2^{20}$ bytes (= 4MB), which surely fits in the cache memory, where an access operation takes $O(1)$ time under many reasonable models. Indeed, we showed that the k-mer rank-retrieval time for such a lookup table is comparable to that of the computationally trivial XOR hash function. For larger k, one can generate a minimizer for $k = 20$ and use Extend-k multiple times, keeping the size of the lookup table within the same 4MB.

Our study raises several open questions. First, for which (σ, k, w) combinations minimizers are optimal among all forward schemes? Second, what is the complexity of the minimum-density problem, and can we generate minimum-density minimizers efficiently? Third, can we prove any guarantees for GreedyE, e.g. whether greedy is *always* better than random? Fourth, is there an algorithm that returns the rank of a k-mer in the order generated by GM-expected or GM-particular without storing the order explicitly?

There are several directions for future work. First, we plan to plug our new minimizers in schemes, such as strobemers [16], to improve the density and study the impact on other metrics, such as conservation. Second, we plan to extend GreedyMini to complete any partially constructed order to a UHS order, such as an order defined on a double-decycling set. Third, we plan to incorporate our new minimizers in existing HTS algorithms and data structures and study the impact on their performance in terms of runtime and memory usage.

6. Code availability

The toolkit and its source code are publicly available at: https://github.com/OrensteinLab/GreedyMini.

7. Competing interests

No competing interest is declared.

8. Author contributions statement

S.G., M.K, and A.S. performed the theoretical analysis. I.T. implemented and benchmarked the toolkit. S.G., A.S., and Y.O. led the project. All authors wrote the manuscript.

9. Acknowledgments

Shay Golan is supported by Israel Science Foundation grant no. 810/21. Ido Tziony acknowledges the cloud-computing credit support of the Israel Data Science and AI Initiative. Arseny Shur is supported by the ERC grant MPM no. 683064 under the EU's Horizon 2020 Research and Innovation Programme and by the State of Israel through the Center for Absorption in Science of the Ministry of Aliyah and Immigration.

References

1. Camille Moeckel, Manvita Mareboina, Maxwell A. Konnaris, Candace S.Y. Chan, Ioannis Mouratidis, Austin Montgomery, Nikol Chantzi, Georgios A. Pavlopoulos, and Ilias Georgakopoulos-Soares. A survey of k-mer methods and applications in bioinformatics. *Computational and Structural Biotechnology Journal*, 23:2289–2303, 2024.

2. Hongyu Zheng, Guillaume Marçais, and Carl Kingsford. Creating and using minimizer sketches in computational genomics. *Journal of Computational Biology*, 30(12):1251–1276, 2023.
3. Malick Ndiaye, Silvia Prieto-Baños, Lucy M Fitzgerald, Ali Yazdizadeh Kharrazi, Sergey Oreshkov, Christophe Dessimoz, Fritz J Sedlazeck, Natasha Glover, and Sina Majidian. When less is more: sketching with minimizers in genomics. *Genome Biology*, 25(1):270, 2024.
4. 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, 2004.
5. 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, 2017.
6. Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Carl Kingsford. Compact universal k-mer hitting sets. In *Algorithms in Bioinformatics: 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22-24, 2016. Proceedings 16*, pages 257–268. Springer, 2016.
7. Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Car Kingsford. Designing small universal k-mer hitting sets for improved analysis of high-throughput sequencing. *PLoS Computational Biology*, 13(10):e1005777, 2017.
8. Barış Ekim, Bonnie Berger, and Yaron Orenstein. A randomized parallel algorithm for efficiently finding near-optimal universal hitting sets. In *International Conference on Research in Computational Molecular Biology*, pages 37–53. Springer, 2020.
9. 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*, 33(7):1154–1161, 2023.
10. Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de Bruijn graphs. *Journal of Computational Biology*, 22(5):336–352, 2015.
11. Minh Hoang, Hongyu Zheng, and Carl Kingsford. Differentiable learning of sequence-specific minimizer schemes with DeepMinimizer. *Journal of Computational Biology*, 29(12):1288–1304, 2022.
12. 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*, 41(1):btae736, 2025.
13. Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Improved design and analysis of practical minimizers. *Bioinformatics*, 36(Supplement_1):i119–i127, 2020.
14. Robert Edgar. Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences. *PeerJ*, 9:e10805, 2021.
15. Ragnar Groot Koerkamp and Giulio Ermanno Pibiri. The mod-minimizer: A simple and efficient sampling algorithm for long k-mers. In *24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*, volume 312, pages 11:1–11:23. Springer, 2024.
16. Kristoffer Sahlin. Effective sequence similarity detection with strobemers. *Genome Research*, 31(11):2080–2094, 2021.
17. Guillaume Marçais, Dan DeBlasio, and Carl Kingsford. Asymptotically optimal minimizers schemes. *Bioinformatics*, 34(13):i13–i22, 2018.
18. Harold Fredricksen and James Maiorana. Necklaces of beads in k colors and k-ary de bruijn sequences. *Discret. Math.*, 23(3):207–210, 1978.
19. Joe Sawada, Aaron Williams, and Dennis Wong. A simple shift rule for k-ary de bruijn sequences. *Discret. Math.*, 340(3):524–531, 2017.
20. Ragnar Groot Koerkamp, Daniel Liu, and Giulio Ermanno Pibiri. The open-closed mod-minimizer algorithm. *bioRxiv*, 2024.
21. Ragnar Groot Koerkamp. minimizers - a minimizer scheme implementation in Rust. `https://github.com/RagnarGrootKoerkamp/minimizers`, 2023. Accessed: 2024-10-04.
22. Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Sequence-specific minimizers via polar sets. *Bioinformatics*, 37(Supplement 1):i187–i195, 2021.
23. Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15:1–12, 2014.
24. Derrick E. Wood, Jennifer Lu, and Ben Langmead. Improved metagenomic analysis with Kraken 2. *Genome Biology*, 20(1):257, 2019.
25. Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.
26. Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. Kmc 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 05 2017.
27. Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38(Supplement 1):i185–i194, 06 2022.
28. Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 03 2016.
29. Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.
30. Francesco Andreace et al. Metaprob 2: Metagenomic reads binning based on assembly using minimizers and k-mers statistics. *Journal of Computational Biology: A Journal of Computational Molecular Cell Biology*, 28(11):1052–1062, 2021.
31. Jouni Sirén, Jean Monlong, Xiaowei Chang, Adam M. Novak, Jordan M. Eizenga, Cole Markello, Jonas Aagaard Sibbesen, Glenn Hickey, Pei-Chun Chang, Adam Carroll, Namrata Gupta, Stacey Gabriel, Tom W. Blackwell, Aakrosh Ratan, Kent D. Taylor, Stephen S. Rich, Jerome I. Rotter, David Haussler, Erik Garrison, and Benedict Paten. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science*, 374(6574):abg8871, 2021.
32. Mikko Rautiainen and Tobias Marschall. GraphAligner: rapid and versatile sequence-to-graph alignment. *Genome Biology*, 21(1):253, 2020.

Supplementary Notes

S1. Preliminaries

S1.1. Strings as integers

In our algorithms, we view k-mers and explicitly stored windows as **integers written in σ-ary** (padded with leading zeroes to the required length). As a result, many string operations, like extracting a k-mer in a given position, become unit-cost integer operations. In Algorithms S7 and S8, the integer operations are made explicit; the shift operation $v \gg_\sigma n$ deletes the last n digits (characters) of v.

S1.2. Tries and reversed tries

Given a set U of strings, its *trie* (or *prefix tree*) $\mathcal{T}_U$ is defined as follows. The set of nodes is the set of all prefixes of the strings in U; the set of edges is the set of all ordered pairs (z, za) of nodes, where $a \in \Sigma$. Similarly, the *reversed trie* $\mathcal{T}_U^{\leftarrow}$ has all *suffixes* of the strings in U as nodes and all ordered pairs of nodes of the form (z, az) as edges.

We use tries and reversed tries of sets of $(k + w)$-windows, computing node characteristics during a depth-first search (DFS) without explicitly storing the whole tree. We assume that each node z of a trie (respectively, reversed trie) with $|z| \geq k$ is labeled by its k-suffix (respectively, k-prefix).

S1.3. Relation between proofs of theorems

The algorithms proving Theorems 1, 2, and 5 share common ideas. We present these ideas from the simplest to the hardest case, so we first prove Theorem 5, then Theorem 2, and finally Theorem 1. After that, we prove Theorems 3 and 4, which also share common ideas in the proofs.

S2. Proof of Theorem 5

To calculate the density $d_{(\rho,w)}$, we count the $(w+k)$-windows charged by ρ and divide the result by σ^{w+k}. We focus on counting charged windows, doing a separate count for windows that are *prefix-charged* (the minimal k-mer is a prefix) and *suffix-charged* (the minimal k-mer is a unique suffix). Two different approaches, described below, result in the algorithms DenDFS and DenDP, reaching the complexity stated in Theorem 5 (a) and (b), respectively.

S2.1. Algorithm DenDFS

Let U be the set of all windows with some k-prefix u. The trie $\mathcal{T}_U$ of U consists of the path from the root to u and a complete σ-ary tree of depth w, rooted at u. In order to count prefix-charged windows with the prefix u, we run a recursive DFS on this complete subtree by calling $\mathsf{DFSp}(u, u, 0, \mathsf{rank}_\rho(u))$ (Algorithm S1).

With each node z we associate the minimum rank r_z of a k-mer occurring in z, i.e., $r_z = \min\{\mathsf{rank}_\rho(u') \mid u' \text{ occurs in } z\}$. If z is internal, it recursively calls all its children; the minimum rank for a child za is computed as $\min\{r_z, \mathsf{rank}_\rho(z_a)\}$, where z_a is the k-suffix of za. After that, z returns the number of prefix-charged windows in its subtree, summing up the values returned by its children. If z is a leaf, i.e., a window in U, it returns 1 if it is prefix-charged (which means exactly $r_z = \mathsf{rank}_\rho(u)$) and 0 otherwise.

Algorithm S1 : function DFSp for recursive count of prefix-charged $(w+k)$-windows

Require: k-prefix u, current node z, depth d, minimum rank r; global $\sigma, k, w, \mathsf{rank}_\rho$
Ensure: number of prefix-charged windows with the prefix z for the minimizer (ρ, w)

```
if d < w then                                          ▷ z is internal
    c ← 0                                              ▷ counter of charged windows
    for a ∈ Σ do                                       ▷ call children
        z_a ← k-suffix of za
        c ← c + DFSp(u, za, d + 1, min{r, rank_ρ(z_a)})
    end for
    return c
else                                                   ▷ z is a leaf
    return [r = rank_ρ(u)]
end if
```

The call $\mathsf{DFSp}(u, u, 0, \mathsf{rank}_\rho(u))$ spends $O(\sigma)$ time per internal node of $\mathcal{T}_U$ and $O(1)$ time per leaf, which is $O(\sigma^w)$ in total. It stores $O(1)$ numbers per active recursive call, which means $O(w)$ space in total.

We count suffix-charged windows with the suffix u in a symmetric way, using a recursive function DFSs on the reversed trie $\mathcal{T}_{U'}^{\leftarrow}$, where U' consists of all $(w + k)$-windows with the suffix u. The difference with DFSp is that a leaf z returns 1 if z contains, apart from the suffix u, no k-mers of rank smaller than or **equal to** $\mathsf{rank}_\rho(u)$. To ensure correct processing, we set the value returned by a leaf z to $[r_z = \mathsf{rank}_\rho(u) + 1]$ and start the search by calling $\mathsf{DFSs}(u, u, 0, \mathsf{rank}_\rho(u) + 1)$. This call obviously satisfies the complexity bounds computed above for DFSp.

Algorithm DenDFS calls $\mathsf{DFSp}(u, u, 0, \mathsf{rank}_\rho(u))$ and $\mathsf{DFSs}(u, u, 0, \mathsf{rank}_\rho(u) + 1)$ for each ranked k-mer u and sums up the results to get the number of windows charged by ρ. The algorithm spends $O(|H|\sigma^w)$ time and $O(w)$ space, as required by statement (a).

S2.2. Algorithm DenDP

Recall that *order-k deBrujin graph over* Σ is a directed σ-regular graph, having all σ-ary k-mers as nodes and all pairs (au, ub), where $u \in \Sigma^{k-1}$, $a, b \in \Sigma$, as edges. If an edge (au, ub) is labeled by b, the graph becomes a deterministic finite automaton $\mathcal{B}$. We view $\mathcal{B}$ as a *transition table* with rows indexed by k-mers and columns indexed by letters; the entry $\mathcal{B}[u, a]$ contains the successor of u by the letter a.

We complete the UHS order ρ to a linear order $\bar{\rho}$, assigning the remaining ranks lexicographically. Then we replace all elements and all row indices in $\mathcal{B}$ with their $\bar{\rho}$-ranks, and sort the rows. The resulting table is referred to as $\mathcal{B}_\rho$.

To count $(w+k)$-windows charged due to some k-mer u, we proceed by dynamic programming. Let pref_r be a two-dimensional table such that $\mathsf{pref}_r[i, j]$ is the number of strings of length $k+j$ having the k-prefix of rank r, the k-suffix of rank i, and all other k-mers of rank at least r. Then the number of $(w+k)$-windows prefix-charged due to rank-r k-mer is $\sum_{i \geq r} \mathsf{pref}_r[i, w]$. Similarly, let suff_r be a two-dimensional table such that $\mathsf{suff}_r[i, j]$ is the number of strings of length $k+j$ having the k-suffix of rank i and all other k-mers of rank greater than r. Then the number of $(w+k)$-windows suffix-charged due to rank-r k-mer is $\mathsf{suff}_r[r, w]$.

Note that $\mathsf{pref}_r[i, 0] = [i = r]$, $\mathsf{suff}_r[i, 0] = 1$, and the DP rules for pref_r and suff_r are almost the same. Namely, $\mathsf{pref}_r[i, j+1] = \sum_\ell \mathsf{pref}_r[\ell, j]$, where the summation is over all ranks $\ell \geq r$ such that the k-mer of rank i is a successor of the k-mer of rank ℓ. The rule is computed by the function DPp (Algorithm S2), which uses the transition table $\mathcal{B}_\rho$ to propagate the counts from the j'th column to the $(j+1)$th column along the edges of the deBrujin graph. The DP rule for suffixes looks the same: $\mathsf{suff}_r[i, j+1] = \sum_\ell \mathsf{suff}_r[\ell, j]$; the only difference is that the inequality for ℓ is strict: $\ell > r$. This rule is computed by the function DPs (Algorithm S3).

Algorithm S2 : function DPp for count of prefix-charged $(w+k)$-windows with a prefix of a given rank

Require: rank r; global $\sigma, k, w, \mathcal{B}_\rho$
Ensure: number of prefix-charged windows with the prefix of rank r for the minimizer (ρ, w)

```
pref[0..σ^k − 1, 0..w] ← zero matrix
pref[r, 0] ← 1
for j ← 1, ..., w do
    for i ← r, ..., σ^k − 1 do
        for a ∈ Σ do                                   ▷ propagate the current count of i to its successors
            pref[B_ρ[i, a], j] ← pref[B_ρ[i, a], j] + pref[i, j − 1]
        end for
    end for
end for
c ← 0
for i ← r, ..., σ^k − 1 do
    c ← c + pref[i, w]                                 ▷ summing prefix-charged windows over all suffixes
end for
return c
```

Algorithm S3 : function DPs for count of suffix-charged $(w+k)$-windows with a suffix of a given rank

Require: rank r; global $\sigma, k, w, \mathcal{B}_\rho$
Ensure: number of suffix-charged windows with the suffix of rank r for the minimizer (ρ, w)

```
suff[0..σ^k − 1, 1..w] ← zero matrix
suff[0..σ^k − 1, 0] ←  column of 1's
for j ← 1, ..., w do
    for i ← r + 1, ..., σ^k − 1 do
        for a ∈ Σ do                                   ▷ propagate the current count of i to its successors
            suff[B_ρ[i, a], j] ← suff[B_ρ[i, a], j] + suff[i, j − 1]
        end for
    end for
end for
return suff[r, w]                                      ▷ all suffix-charged windows have the same suffix (of rank r)
```

Algorithm S2 works in $O(w\sigma^k)$ time (recall that we assume σ to be a constant) and uses $O(\sigma^k)$ space, as just two columns of the DP table are stored at every moment. The same bounds apply for counting suffix-charged windows.

The algorithm DenDP creates the table $\mathcal{B}_\rho$ and then calls DPp(r) and DPs(r) for each rank r assigned in ρ. As $\mathcal{B}_\rho$ can be trivially computed from rank_ρ in $O(\sigma^k)$ time and space, DenDP works in $O(|H| w\sigma^k)$ time and $O(\sigma^k)$ space, as required in statement (b). Theorem 5 is proved.

Remark S1 *An additional feature of the dynamic programming algorithm* DenDP *is that while computing* $d_{(\rho,w)}$ *it can, within the same time and space bounds, report the densities of* ***all*** *minimizers* (ρ, w') *with* $w' \leq w$.

S3. Proof of Theorem 2

Let ρ be a UHS order for w, H be its UHS, and r be a rank such that both r and $r{+}1$ are assigned in ρ. Let ρ' be the order obtained from ρ by swapping k-mers u and u' such that $\mathsf{rank}_\rho[u] = r$, $\mathsf{rank}_\rho[u'] = r+1$. We define the *cost function* $\mathsf{cost}(\rho, w, r) = \sigma^{k+w}(d_{(\rho',w)} - d_{(\rho,w)})$, equal to the difference between the numbers of $(w+k)$-windows charged by ρ' and by ρ. We focus on the efficient computation of $\mathsf{cost}(\rho, w, r)$, since all other operations in the function Swap are straightforward (Algorithm S4). Similar to Theorem 5, we present two algorithms computing the cost function: the DFS-based algorithm and the DP-based algorithm are used to prove the statements (a) and (b) of Theorem 2, respectively. Both algorithms analyse the difference between ρ and ρ', avoiding direct computation of $d_{(\rho',w)}$.

Algorithm S4 : function Swap for swapping two k-mers with consecutive ranks

Require: array $\mathsf{rank} = \mathsf{rank}_\rho$ for a UHS order ρ for w, rank r; global σ, k, w
Ensure: array rank after an attempted swap of ranks r and $r{+}1$

```
u ← ρ(r); u' ← ρ(r+1)
c ← cost(ρ, w, r)
if c < 0 then
    swap the elements rank[u] and rank[u']
else if c = 0 then
    swap the elements rank[u] and rank[u'] with probability 0.5
end if
return rank
```

S3.1. Algorithm SwapDFS

In this section, compute $\mathsf{cost}(\rho, w, r)$ by considering only those $(w{+}k)$-windows that could have different status (free/charged) for ρ and ρ'. If a window v changed its status, then its minimum-rank k-mer in ρ is u and its minimum-rank k-mer in ρ' is u'. Hence v contains both u and u'; moreover, either u or u' is a prefix and/or a suffix of v. Therefore, it is sufficient to check four sets of windows: those with prefix u, prefix u', suffix u, and suffix u'. The number of windows in these sets is $O(\sigma^w)$.

Let U be the set of all windows with the prefix u and let $v \in U$. If v contains a k-mer of rank $< r$, then it has the same status for both ρ and ρ'. Otherwise, it is charged by ρ; this charged window is free for ρ' if it contains u', but not as a unique suffix. Respectively, it suffices to count the windows that are charged by ρ and free for ρ', as the opposite change of status is impossible. Similar to the proof of Theorem 5, we make use of the trie $\mathcal{T}_U$ of U. It consists of a path from the root to u, followed by a complete σ-ary tree of depth w. We additionally assign to each internal node z a flag indicating whether u' occurs in z and run a DFS on the complete subtree, calling the function $\mathsf{pr_cost}(\mathsf{rank}_\rho, r, u, u', u, 0, \mathsf{False})$ (Algorithm S5).

During the DFS, each node returns the number of $(w+k)$-windows in its subtree that are charged by ρ and free for ρ'. For a node z with the label z', if $\mathsf{rank}[z'] < r$, we return 0. Otherwise, the processing depends on the type of the node. Let z be internal. If $z' = u'$, we set the flag; otherwise, we copy its value from the parent of z. After that, we recursively call all children of z and return the sum of the values they returned. Now let z be a leaf. It returns 1 if the flag of its parent is set and 0 otherwise. Clearly, the root u of the subtree returns the required value. The processing time is $O(\sigma^w)$ and the space is $O(w)$, as we store only $O(1)$ numbers per each of $O(w)$ active recursive calls. We then process the set U' of windows with the prefix u' in the same way as U, calling $\mathsf{pr_cost}(\mathsf{rank}_\rho, r, u', u, u', 0, \mathsf{False})$ to count the windows that are charged by ρ' and free for ρ.

Algorithm S5 : function pr_cost for recursive count of status-changing k-mers

Require: array rank, rank r, k-mers u, u' to swap, current node with k-suffix z and depth d, flag f (set if u' is already seen); global σ, k, w
Ensure: number c of windows with prefix u, in the subtree of the current node, that become free if u and u' swapped

```
if rank[z] < r then                                   ▷ status of windows containing z is independent of u, u'
    return 0
end if
if d = w then                                         ▷ current node is a leaf (window)
    return [f]
else                                                  ▷ current node is internal
    f ← f ∨ (z = u')
    c ← 0
    for a ∈ Σ do                                      ▷ recursive call for children
        c ← c + pr_cost(rank, r, u, u', (σ · z) mod σ^k + a, d + 1, f)
    end for
end if
return c
```

Consider the set $\bar{U}$ of the windows with the suffix u. The options for $v \in \bar{U}$ are the following. If v contains a k-mer of rank $< r$ or it does not contain u', then it has the same status in ρ and ρ'. If v has the prefix u or u', we ignore it to avoid double count. Otherwise, v is free for ρ' and charged by ρ if and only if it has a single occurrence of u. We process $\bar{U}$ similar to U, counting the windows that are charged by ρ and free for ρ', but in the **reversed** trie $\mathcal{T}^{\leftarrow}_{\bar{U}}$. We use a DFS over the complete subtree of $\mathcal{T}^{\leftarrow}_{\bar{U}}$ rooted at u and adjust the processing of a node according to the conditions listed above, getting the function sf_cost (Algorithm S6). It has the same space and time bounds as pr_cost, and the set $\bar{U}'$ of the windows with the suffix u' is processed in exactly the same way as $\bar{U}$. Finally, $\mathsf{cost}(\rho, w, r) = \mathsf{pr_cost}(\mathsf{rank}_\rho, r, u', u, u', 0, \mathsf{False}) + \mathsf{sf_cost}(\mathsf{rank}_\rho, r, u', u, u', 0, \mathsf{False}) - \mathsf{pr_cost}(\mathsf{rank}_\rho, r, u, u', u, 0, \mathsf{False}) - \mathsf{sf_cost}(\mathsf{rank}_\rho, r, u, u', u, 0, \mathsf{False})$ is computed in $O(\sigma^w)$ time and $O(w)$ space. Adding $O(\sigma^k)$ time for finding the k-mers u and u' (Algorithm S4, line 1), we get the complexity bounds of Theorem 2(a).

Algorithm S6 : function sf_cost for recursive count of status-changing k-mers

Require: array rank, rank r, k-mers u, u' to swap, current node with k-prefix z and depth d, flag f (set if u' is already seen); global σ, k, w

Ensure: number c of windows with unique suffix u, in the subtree of the current node, that become free if u and u' swapped

```
 1: if (rank[z] < i) ∨ ((d > 0) ∧ (z = u)) then        ▷ status of windows is independent of u, u'
 2:     return 0
 3: end if
 4: if d = w then                                       ▷ current node is a leaf (window)
 5:     return [f ∧ (z ≠ u')]   ▷ windows with prefix u' are processed using Alg. S5 and hence ignored here
 6: else                                                ▷ current node is internal
 7:     f ← f ∨ (z = u')
 8:     c ← 0
 9:     for a ∈ Σ do                                    ▷ recursive call for children
10:         c ← c + sf_cost(rank, i, u, u', (aσ^k + z) ≫_σ 1, d + 1, f)
11:     end for
12: end if
13: return c
```

Algorithm SwapDFS runs multiple iterations of local search. Each iteration calls $\mathsf{Swap}(\rho, w, r)$ for the current UHS order ρ and a randomly chosen rank r; within this call, the cost function is computed by the depth-first search as described above.

S3.2. Algorithm SwapDP

From Section S2, we use the function DPp counting $(w+k)$-windows prefix-charged due to a k-mer of a given rank (Algorithm S2), and its counterpart DPs for suffix-charged windows (Algorithm S3). Let $c(\rho, w, i)$ be the number of $(w+k)$-windows charged by ρ due to the rank-i k-mer. Then $\mathsf{cost}(\rho, w, r) = c(\rho', w, r) + c(\rho', w, r+1) - c(\rho, w, r) - c(\rho, w, r+1)$. Accordingly, we compute $c(\rho, w, r) = \mathsf{DPp}(r) + \mathsf{DPs}(r)$ and $c(\rho, w, r+1) = \mathsf{DPp}(r+1) + \mathsf{DPs}(r+1)$ using the transition table $\mathcal{B}_\rho$, then edit $\mathcal{B}_\rho$ to get $\mathcal{B}_{\rho'}$, and compute $c(\rho', w, r)$, $c(\rho', w, r+1)$ in the same way, but with the table $\mathcal{B}_{\rho'}$. Note that given $\mathcal{B}_\rho$, ρ, and rank_ρ, the transition table $\mathcal{B}_{\rho'}$ can be computed in constant time. Indeed, we take $u = \rho(r)$, compute all its predecessors by $O(1)$ arithmetic operations, extract their ranks from rank_ρ and replace r with $r+1$ in the rows of $\mathcal{B}_\rho$ corresponding to these ranks. Then we do the same with $u' = \rho(r+1)$, and replace $r+1$ with r in the obtained rows. Finally we swap the rows r and $r+1$, getting $\mathcal{B}_{\rho'}$.

In this way, $\mathsf{cost}(\rho, w, r)$ is computed within the complexity bounds of Algorithm S2, i.e., in $O(w\sigma^k)$ time and $O(\sigma^k)$ space. Then we immediately have statement (b) of Theorem 2.

Algorithm SwapDP runs multiple iterations of local search. Before the first iteration, it spends $O(\sigma^k)$ time to compute $\mathcal{B}_{\rho_0}$ for the input UHS order ρ_0. Each iteration calls $\mathsf{Swap}(\rho, w, r)$ for the current UHS order ρ and a randomly chosen rank r; within this call, the cost function is computed by the dynamic programming algorithms DPp and DPs as described above. The changes in the transition table made during this computation are retained if the swap was successful and reversed otherwise.

S4. Proof of Theorem 1

As with Theorems 2 and 5, both DFS-based and DP-based solutions are possible. For an unranked k-mer u, counting $\mathsf{score}(u) = |Y_u|/|X_u|$ can be done in $O(w\sigma^k)$ time by variants of the functions DPp (Algorithm S2) and DPs (Algorithm S3). However, this means $O(w\sigma^{2k})$ time per iteration, which is rather inefficient. Because of this, we stick to the DFS solution.

We maintain arrays X, Y, and rank, each of size σ^k. For every k-mer u, the counters $\mathsf{X}[u]$ and $\mathsf{Y}[u]$ store the current values of $|X_u|$ and $|Y_u|$, respectively; rank is used to store the assigned ranks over all k-mers. To initialize X and Y it suffices to loop through all windows, incrementing the counters of all k-mers occurring in the current window. This can be done in $O(w\sigma^{w+k})$ time in a trivial way. Then at most σ^k iterations follow, each consisting of two steps: choosing the k-mer to assign rank to and updating the arrays X, Y (for convenience, the rank is assigned *after* the updates). Computing the pool of low-scored k-mers and choosing a random k-mer from it can be done in time proportional to the size of X and Y, i.e., $O(\sigma^k)$.

Assume that the k-mer u is chosen and consider the update step (Algorithm S7). For every $i \in [0, w+1)$, exactly σ^w windows contain u at position i. Hence, the number of windows containing u is $O(w\sigma^w)$, and to fit into the time bound we need to spend amortized $O(1)$ time per such window while skipping all windows that do not contain u. We group windows containing u in u-*blocks*. A u-block is a set of all windows v sharing the same prefix pu, where p is followed by the leftmost u in v. The number of u-blocks

with the leftmost u starting at position i equals the number of options for p, which is at most σ^i. Hence, the total number of u-blocks is at most $\sum_{i=0}^{w} \sigma^i = O(\sigma^w)$.

Algorithm S7 : function update_counts

Require: Arrays X, Y, rank, k-mer u; global σ, k, w
Ensure: Arrays X', Y' updated from X and Y

```
v, d ← nextblock(0, u)                                   ▷ compute the first u-block (Algorithm S8)
while v < ∞ do                                           ▷ current block exists
    seen ← ∅
    for i ← 0 to d − 1 do                                ▷ processing k-mers starting to the left of u
        z ← (v ≫_σ (w−i)) mod σ^k                        ▷ k-mer starting at position i
        if rank[z] < ∞ then                              ▷ no live windows in this block, move to next
            v, d ← nextblock((v ≫_σ (w−i)) + 1)) · σ^(w−i), u); break   ▷ compute the next u-block (Algorithm S8)
        end if
        seen ← seen ∪ {z}
    end for
    L, X′, Y′ ← DFS_count_and_update(u, d, seen, X, Y)   ▷ Algorithm S9
    for z ∈ seen do                                      ▷ subtract the number of live windows in the current block
        X′[z] ← X′[z] − L
    end for
    Y′[v ≫_σ w] ← Y′[v ≫_σ w] − L                        ▷ for the prefix of v live windows are charged
    v, d ← nextblock((v ≫_σ (w−d)) + 1)) · σ^(w−i), u)   ▷ move to the next block
end while
return X′, Y′
```

We process one block at a time, navigating between blocks with the help of an auxiliary function nextblock(v, u) (Algorithm S8): it returns the smallest window v' that is greater than the window v and contains u, or null if no such window exists. In other words, nextblock(v, u) returns the first window of the next u-block. To implement nextblock(v, u), we compute, for every $i \in [0, w+1)$, the smallest window v_i (if any) that is greater than v and contains u at position i. Computing v_i from v, u, i requires a constant number of arithmetic operations. Hence, nextblock(v, u) can be computed in $O(w)$ time.

Algorithm S8 : function nextblock

Require: window v, k-mer u; global σ, k, w
Ensure: smallest window $v' \geq v$ that contains u, leftmost position d of u in v'

```
v′ ← ∞; d ← ∞                                            ▷ no window found yet
for i ← 0 to w do
    z ← (v mod σ^(w+k−i)) ≫_σ (w − i)                    ▷ the k-mer starting at position i in m
    if z < u then                                        ▷ replace u with z and the rest with 0's
        x ← ((v ≫_σ (w−i)) + u − z) · σ^(w−i)
    else if z > u then                                   ▷ same + add 1 to the digit preceding u
        x ← ((v ≫_σ (w−i)) + σ^k + u − z) · σ^(w−i)
    else
        return v, i                                      ▷ v contains u, the leftmost occurrence is at i
    end if
    if x < min{v′, σ^(w+k)} then                         ▷ smaller window found
        v′ ← x; d ← i
    end if
end for
return v′, d
```

Consider processing a u-block U (one iteration of the **while** cycle in Algorithm S7). The goal is to count, for each unranked k-mer u' occurring in at least one window of U, (i) the number of live windows from U containing u' and (ii) the number of live windows from U containing u' as a prefix or as a unique suffix. These numbers are then subtracted from $\mathsf{X}[u']$ and $\mathsf{Y}[u']$ respectively; in particular, $\mathsf{X}[u]$ and $\mathsf{Y}[u]$ become zeroes. Let pu be the common prefix of all windows in U, $i = |p|$, and let $\mathcal{T}_U$ be the trie of U. Each node of $\mathcal{T}_U$ is identified with a prefix of some windows from U, and leaves are identified with the windows themselves. The trie consists of a single path from the root to the node pu followed by a complete σ-ary tree of depth $w - i$ below pu. Recall that each node of this complete tree is labeled with its k-suffix.

We first check all k-mers in pu except for u; if any of them has an assigned rank, then U contains no live windows, so the processing is finished (since all numbers to be subtracted from the cells of X and Y are zeroes). Otherwise, we put all of them in

Algorithm S9 : function DFS_count_and_update

Require: k-mer z, depth d, set seen of occurred k-mers, arrays X, Y; global σ, k, w
Ensure: number L of live windows below z, arrays X', Y' updated from X and Y

```
if X[z] = 0 then                                   ▷ no live windows contain z (includes the case rank(z) < ∞)
    return 0, X, Y
else
    X' ← X; Y' ← Y
    f ← z ∉ seen                                   ▷ set flag f if z is a unique suffix of the current node
    if d = w then                                  ▷ current node is a leaf (live window)
        if f then                                  ▷ z is a unique suffix, decrement both X'[z], Y'[z]
            X'[z] ← X'[z] − 1; Y'[z] ← Y'[z] − 1
        end if
        return 1, X', Y'
    else                                           ▷ current node is internal
        if f then
            seen ← seen ∪ {z}                      ▷ add z to occurred k-mers
        end if
        ℓ ← 0                                      ▷ initialize counter of live windows
        for i ← 0 to σ − 1 do                      ▷ call DFS for children
            z' ← (σ · z) mod σ^k + i
            c, X', Y' ← DFS_count_and_update(z', d + 1, seen, X', Y')
            ℓ ← ℓ + c
        end for
        if f then                                  ▷ subtract live windows from X'[z]
            X'[z] ← X'[z] − ℓ; seen ← seen \ {z}
        end if
        if d = 0 then                              ▷ z is a prefix, decrease Y'[z]
            Y'[z] ← Y'[z] − ℓ
        end if
        return ℓ, X', Y'
    end if
end if
```

a set seen of occurred k-mers, associate seen with the node pu of $\mathcal{T}_U$, and run a recursive DFS (Algorithm S9) on the complete subtree of $\mathcal{T}_U$.

When the DFS reaches a node z with the label z', we first check if z' has an assigned rank. If yes, there are no live windows in the subtree of z, so this subtree is skipped and z returns 0. Otherwise, we set a boolean flag if $z' \in$ seen. After that, if z is an internal node, we insert z' into seen if it was not there, and recursively call the children of z. When all children return values, we compute and return their sum s. If the boolean flag was not set, then before returning we delete z' from seen and subtract s from $\mathsf{X}[z']$ (if the flag *was set*, the subtraction will be done when processing the topmost occurrence of z' among ancestors of z). Finally, if z is a leaf, we just return 1; if the flag was not set, before returning we subtract 1 from both $\mathsf{X}[z']$ and $\mathsf{Y}[z']$, since z' is a unique suffix of a live window z.

When the DFS is finished, the root pu of the complete subtree returns the number $\hat{s}$ of live windows in U. Now it remains to subtract $\hat{s}$ from $\mathsf{X}[u']$ for each k-mer u' in pu (excluding u, which is already processed) and also from $\mathsf{Y}[u'']$ for the k-prefix u'' of pu. Thus, we reached the claimed goal.

When all u-blocks are processed, each value $\mathsf{X}[u']$ (respectively, $\mathsf{Y}[u']$) was decreased by the number of live windows that contain u and also u' (respectively, also u' as a prefix or as a unique suffix). Since the live windows containing u are exactly those losing the state of "live" after the current iteration, the arrays X and Y are updated correctly.

It remains to check the time and space complexity. Processing a u-block U requires $O(w)$ time for the k-mers in the common prefix pu and $O(1)$ time per node of the complete subtree of the prefix tree $\mathcal{T}_U$. As there are $O(\sigma^w)$ u-blocks and the size of a complete subtree is proportional to the number of leaves in it, each of two processing times sums up to $O(w\sigma^w)$ over all u-blocks. We also make $O(\sigma^w)$ calls to the nextblock function to move to the next u-block after finishing with the current one; as one call costs $O(w)$ time, we arrive at the same bound. Adding $O(\sigma^k)$ time to select the k-mer u and multiplying the result by the number of iterations, we get the claimed time bound.

As for the space complexity, we do not store the subtree in which we run the DFS: the navigation is by arithmetic operations, so at any moment we store the set seen of $O(w)$ size plus $O(1)$ numbers per each of $O(w)$ active calls. Adding the size of the arrays X, Y, and rank, we get the desired space bound. Theorem 1 is proved.

S5. Discussion on σ- and k-extensions

In this section we analyze whether the straightforward density-preserving σ- and k-extensions of a minimizer are minimizers.

S5.1. σ-extension

Let $f = (\rho, w)$ be a minimizer with the parameters (σ, k, w) and $\gamma > 1$ be an integer. Let $\bar{*}$ be any *projection* function from the alphabet $\{0, \ldots, \sigma\gamma - 1\}$ onto $\{0, \ldots, \sigma - 1\}$. The local scheme f_γ with the parameters $(\sigma\gamma, k, w)$ is defined as follows: in every window v, f_γ picks the position chosen by f in the *projection* $\bar{v}$ of v, obtained from v by applying $\bar{*}$ to each letter of v.

Lemma 2 f_γ is not a minimizer.

Proof Suppose that f_γ is a minimizer (π, w). By pigeonhole principle there must be a symbol $s \in \{0, \ldots, \sigma - 1\}$ such that for two distinct $i, j \in \{0, \ldots, \sigma\gamma - 1\}$ we have $s = \bar{i} = \bar{j}$. Consider the k-mer u having the maximal π-rank among k-mers with the projection s^k. Let v be a window with the projection $\bar{v} = s^{w+k-1}$ such that $v[0, k) = u$ and $v[1, k+1) = u' \neq u$. Then f_γ picks the position 0 while $\mathsf{rank}_\pi(u) > \mathsf{rank}_\pi(u')$. Hence $f \neq (\pi, w)$; the contradiction proves that f_γ is not a minimizer. □

S5.2. k-extension

Let $f = (\rho, w)$ be a minimizer with the parameters (σ, k, w) and $k' > k$ be an integer. The local scheme f' with the parameters (σ, k', w) is defined as follows: in every $(w + k' - 1)$-window v, f' picks the position chosen by f in the $(w + k - 1)$-prefix of v.

Lemma 3 If $w > k'$, then f' is not a minimizer.

Proof Suppose that $f' = (\rho', w)$ is a minimizer. Let $u \in \Sigma^k$, $x \in \Sigma^{k'-k}$ be such that $\mathsf{rank}_{\rho'}(ux) = 0$. As a minimizer, f' picks the starting position of ux in every $(w + k' - 1)$-window in which this k'-mer occurs. As $w > k' > k$, every two k-mers appear in some $(w + k' - 1)$-window; hence $\mathsf{rank}_\rho(u) = 0$.

Consider some k'-mer $uy \neq ux$. Since $w > k'$, there exists a $(w + k' - 1)$-window v with the prefix $uyux$. Since $\mathsf{rank}_\rho(u) = 0$, f' by definition picks the position 0 of v, while v contains the k'-mer ux of smaller ρ'-rank. This contradiction proves that f' is not a minimizer. □

If $w \leq k'$, the answer to the question whether f' is a minimizer depends on both w and ρ. In particular, if there exist $(w+k'-1)$-windows v_1 and v_2 such that (i) v_1 begins with the k'-mer ux and contains the k'-mer uy; (ii) v_2 begins with uy and contains ux; (iii) u has the lowest ρ-rank among all k-mers in the $(w + k - 1)$-prefixes of v_1 and v_2, then f' is not a minimizer. Indeed, similar to Lemma 3, f' picks the position 0 in both v_1 and v_2, and thus ux and uy cannot be compared.

The authors of [de Blasio et al., Practical universal k-mer sets for minimizer schemes. ACM-BCB 2019] used the following notion of extension: if ρ and ρ' are orders of k-mers and $(k+1)$-mers respectively, and for every $u, u' \in \Sigma^k$, $a, b \in \Sigma$ the inequality $\mathsf{rank}_\rho(u) < \mathsf{rank}_\rho(u')$ implies $\mathsf{rank}_{\rho'}(ua) < \mathsf{rank}_{\rho'}(u'b)$, then ρ' is an extension of ρ. They claimed, in Theorem 3 of the cited paper, that $d_{(\rho', w)} \leq d_{(\rho, w)}$ for any extension ρ' of ρ. Their proof has a flaw: they claim that if the minimizer (ρ', w) picks position i in some $(w+k)$-window v due to a $(k+1)$-mer ua at this position, then the minimizer (ρ, w) picks the same position i in $v[0, w+k-1)$ due to the k-mer u. But v can contain ub, where $b \neq a$ and $\mathsf{rank}_{\rho'}(ub) > \mathsf{rank}_{\rho'}(ua)$, in a position $j < i$. In this case, (ρ, w) does not pick the position i in $v[0, w + k - 1)$, because ties are broken to the left.

The flawed proof does not imply automatically that Theorem 3 of [de Blasio et al.] fails, but our experiments show that the application of the particular extension defined in Section 3.3 in some cases increases density, so the theorem in the stated form is indeed incorrect.

S6. Proof of Theorem 3

We call a window $v \in (\Sigma \times \Gamma)^{w+k}$ *bad* if it is charged by $\rho \times \tau$ while v_Σ is free for ρ, and *good* if v is free for $\rho \times \tau$ while v_Σ is charged by ρ. Let B and G denote the sets of bad and good windows, respectively. The order $\rho \times \tau$ charges exactly $\gamma^{w+k}c + |B| - |G|$ windows, where c is the number of windows charged by ρ. Consequently, $d_{(\rho\times\tau, w)} = d_{(\rho, w)} + \frac{|B|-|G|}{(\sigma\gamma)^{w+k}}$. Next, let $B^\leftarrow$ and $G^\leftarrow$ be the sets of bad and good windows for the order $\rho^\leftarrow \times \tau$, respectively.

Claim 1 $|B| \leq |G^\leftarrow|$.

Proof of the Claim Let $v \in B$ and let u be the minimum-rank k-mer of v. Then, u_Σ has the minimum ρ-rank among the k-mers of v_Σ. By the definition of bad, v_Σ is not charged by ρ, so u_Σ is neither a prefix nor a unique suffix of v_Σ. Hence, u is a suffix of v, and u_Σ occurs more than once in v_Σ. Moreover, $\mathsf{rank}_\tau(u_\Gamma) < \mathsf{rank}_\tau(z_\Gamma)$ for each other k-mer z of v such that $z_\Sigma = u_\Sigma$. We define the $(w+k)$-window $x = \phi(v)$ as follows: $x_\Sigma = \overleftarrow{v_\Sigma}$, $x_\Gamma = v_\Gamma[i, w+k)v_\Gamma[0, i)$, where i is the starting position of the leftmost occurrence of u_Σ in v_Σ (Figure S1).

By the definition of mirror, the prefix $\overleftarrow{u_\Sigma}$ of x_Σ has the minimum $\rho^\leftarrow$-rank among the k-mers of x_Σ. Then, x_Σ is charged by $\rho^\leftarrow$. Let $z = v[i, i + k)$, $u' = x[0, k)$, $z' = x[w-i, w-i+k)$, and $y = x[w, w + k)$. Since $z_\Sigma = u_\Sigma$ by the choice of i, we have $\mathsf{rank}_\tau(u_\Gamma) < \mathsf{rank}_\tau(z_\Gamma)$. Since $u'_\Sigma = z'_\Sigma = \overleftarrow{u_\Sigma}$, $u'_\Gamma = z_\Gamma$, and $z'_\Gamma = u_\Gamma$, we get $\mathsf{rank}_{\rho^\leftarrow\times\sigma}(z') < \mathsf{rank}_{\rho^\leftarrow\times\sigma}(u')$. Note that $y_\Sigma \neq \overleftarrow{u_\Sigma}$ because u_Σ is not a prefix of v_Σ. As $\overleftarrow{u_\Sigma}$ has the minimum $\rho^\leftarrow$-rank among the k-mers of x_Σ, we have $\mathsf{rank}_{\rho^\leftarrow\times\sigma}(u') < \mathsf{rank}_{\rho^\leftarrow\times\sigma}(y)$. Since z' has smaller rank than both the k-prefix and the k-suffix of x, x is free for $\rho^\leftarrow \times \tau$. Then, $x \in G^\leftarrow$ by the definition of a good window. Hence, ϕ maps B to $G^\leftarrow$.

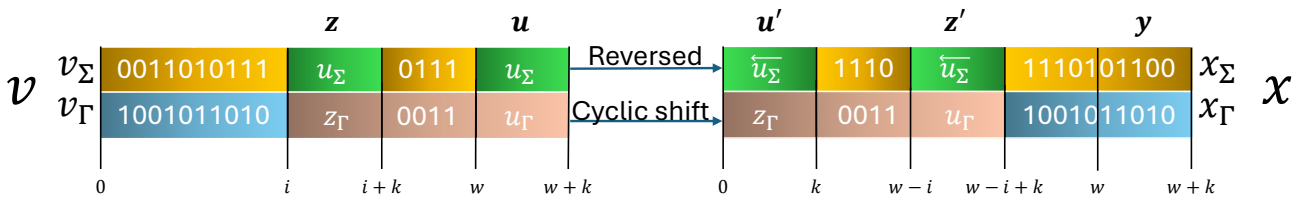

Fig. S1. Example of mapping a bad (for $\rho \times \tau$) window v to a good (for $\rho^{\leftarrow} \times \tau$) window x.

Now let $v, v' \in B$, $v \neq v'$, $x = \phi(v)$, $x' = \phi(v')$. If $v_\Sigma \neq v'_\Sigma$, then $x_\Sigma \neq x'_\Sigma$. If $v_\Sigma = v'_\Sigma$, then x_Γ, x'_Γ are obtained by the same shift of v_Γ and v'_Γ, respectively. Since $v \neq v'$, we have $v_\Gamma \neq v'_\Gamma$, and then $x_\Gamma \neq x'_\Gamma$. Therefore, in both cases $x \neq x'$. Thus, we proved that ϕ is injective. The claim now follows. $\square$

Since $(\rho^{\leftarrow})^{\leftarrow} = \rho$, the claim implies $|B^{\leftarrow}| \leq |G|$. Then

$$d_{(\rho\times\tau,w)} + d_{(\rho^{\leftarrow}\times\tau,w)} = d_{(\rho,w)} + d_{(\rho^{\leftarrow},w)} + \frac{|B|-|G|+|B^{\leftarrow}|-|G^{\leftarrow}|}{(\sigma\gamma)^{w+k}} \leq d_{(\rho,w)} + d_{(\rho^{\leftarrow},w)}.$$

The result now follows by Lemma 1.

S7. Proof of Theorem 4

Similar to the proof of Theorem 3, we define bad and good windows. For $v \in \Sigma^{w+k}$ and $a \in \Sigma$, the window va is bad (good) for ρ_1 if it is charged (not charged) by ρ_1 while v is not charged (charged) by ρ. By B_1 and G_1 we denote the sets of bad and good windows for ρ_1, respectively. Similarly, we define bad and good windows av for ρ_2 and denote their sets by B_2 and G_2, respectively. We prove the analog of the claim from Theorem 3.

Claim 2 $|B_1| \leq |G_2|$ and $|B_2| \leq |G_1|$.

Proof of the Claim Let $va \in B_1$ and let ub be the $(k+1)$-mer in va of minimum ρ_1-rank (here $a, b \in \Sigma$). By the definition of ρ_1, u has the minimum ρ-rank in v. Since v is free for ρ, u is not a prefix of v. Hence, ub is a unique suffix of va; in particular, $b = a$. Then, u is a suffix of v and has other occurrences in v because v is free for ρ. Thus, v contains a $(k+1)$-mer ua', where $a' > a$ as $\mathsf{rank}_{\rho_1}(ua') > \mathsf{rank}_{\rho_1}(ua)$.

The window $\overleftarrow{v}$ is charged by $\rho^{\leftarrow}$ as its minimum-rank k-mer $\overleftarrow{u}$ is its prefix (but not suffix). Consider the window $a\overleftarrow{v}$. Its prefix $a\overleftarrow{u}$ has smaller ρ_2-rank than its $(k+1)$-suffix, as this suffix does not end with $\overleftarrow{u}$. Further, $a\overleftarrow{v}$ contains the $(k+1)$-mer $a'\overleftarrow{u}$ with $a' > a$ and thus $\mathsf{rank}_{\rho_2}(a'\overleftarrow{u}) < \mathsf{rank}_{\rho_2}(a\overleftarrow{u})$. Hence, $a\overleftarrow{v}$ is free for ρ_2. We thus proved $a\overleftarrow{v} \in G_2$. As the reversal of every window from B_1 belongs to G_2, we have $|B_1| \leq |G_2|$. A symmetric argument proves $|B_2| \leq |G_1|$, completing the claim. $\square$

Let $c, c^{\leftarrow}$ be the numbers of windows charged by ρ and $\rho^{\leftarrow}$, respectively. Then, ρ_1 charges $\sigma c + |B_1| - |G_1|$ windows, and ρ_2 charges $\sigma c^{\leftarrow} + |B_2| - |G_2|$ windows. Hence, $d_{(\rho_1,w)} = d_{(\rho,w)} + \frac{|B_1|-|G_1|}{\sigma}$, $d_{(\rho_2,w)} = d_{(\rho^{\leftarrow},w)} + \frac{|B_2|-|G_2|}{\sigma}$. Substituting the inequalities from the claim, we get

$$d_{(\rho_1,w)} + d_{(\rho_2,w)} \leq d_{(\rho,w)} + d_{(\rho^{\leftarrow},w)},$$

implying the theorem in view of Lemma 1.

S8. GreedyMini workflows

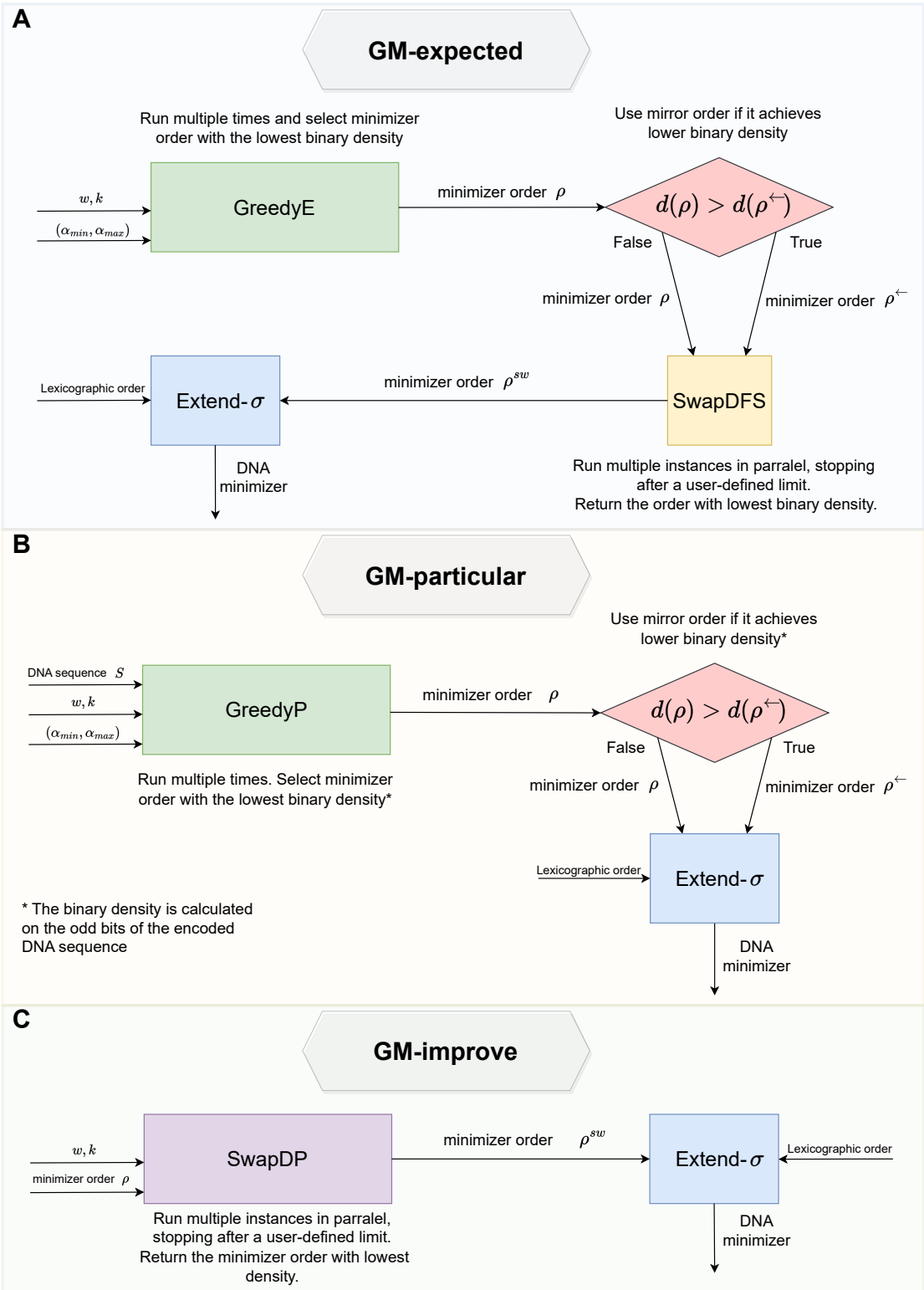

Fig. S2. Workflow illustration of (A) GM-expected pipeline to generate a low-expected-density DNA minimizer, (B) GM-particular pipeline to generate a low-particular-density DNA minimizer, and (C) GM-improve pipeline to improve a (binary) minimizer to larger w values.

S9. Running DeepMinimizer and polar sets

We failed to run DeepMinimizer [11] and polar sets [22] due to errors received when running their code. Both implementations contained unspecified file hierarchies, hard-coded paths, and lacked essential documentation, such as the required dependencies and their versions. For instances where hard-coded paths were present, we replaced them with local paths. After adjusting the paths, the following issues arose:

- **Polar sets**: We ran the following command:

  ```
  python anchor_distributed.py sequence_1M -w 12 --kmin 3
  --kmax 16 -p 10
  ```

  It raised the following error:

  ```
  Traceback (most recent call last):
    File "/home/dsi/tzionyi/polarset/anchor_distributed.py",
    line 18, in <module> assert working_dir is not None
  ```

- **DeepMinimizer**: We ran the following command:

  ```
  python run_scripts.py
  ```

  It raised the following error:

  ```
  RuntimeError: one_hot is only applicable to index tensor.
  ```

S10. Construction runtime and maximum memory usage of GM-expected and GM-particular

We ran GM-expected and GM-particular on a Linux server equipped with 2 × Intel(R) Xeon(R) Gold 6338 CPUs (total cores 64) @ 2.00GHz and 512GB of RAM. We utilized 64 available cores. We report the construction runtime (Supplementary Figure S3A) and maximum memory usage (Supplementary Figure S3B) of our implementation of GM-expected over various combinations of w and k without extending k (Theorem 4).

The measured construction runtimes fit the bound of GreedyE given by Theorem 1 ($O(\sigma^{2k} + w\sigma^{k+w})$). Focusing on $w + k > 18$ to avoid effects of small values, we observe for $k > 2w$ an exponential dependence on k, while for $k \leq 2w$ we observe an exponential dependence on $w + k$. Note that we run the local search by Swap iterations with a time limit according to the runtime of GreedyMini, and hence their runtimes are equal. The maximum memory usage matches that of GreedyE and Swap by Theorems 1 and 2 ($O(\sigma^k + w)$). Note that due to the utilization of various libraries and other computational factors, our implementation of GM-expected has a baseline memory usage, and thus the increase in maximum memory usage is only observed for $k \geq 14$.

In addition, we report the construction runtime (Supplementary Figure S4A) and maximum memory usage (Supplementary Figure S4B) of our implementation of GM-particular. Over a sequence length of 1 million nucleotides, the construction runtime increased exponentially with k for $k > 12$, and increased much more slightly with w. The maximum memory usage increased with w more profoundly than with k.

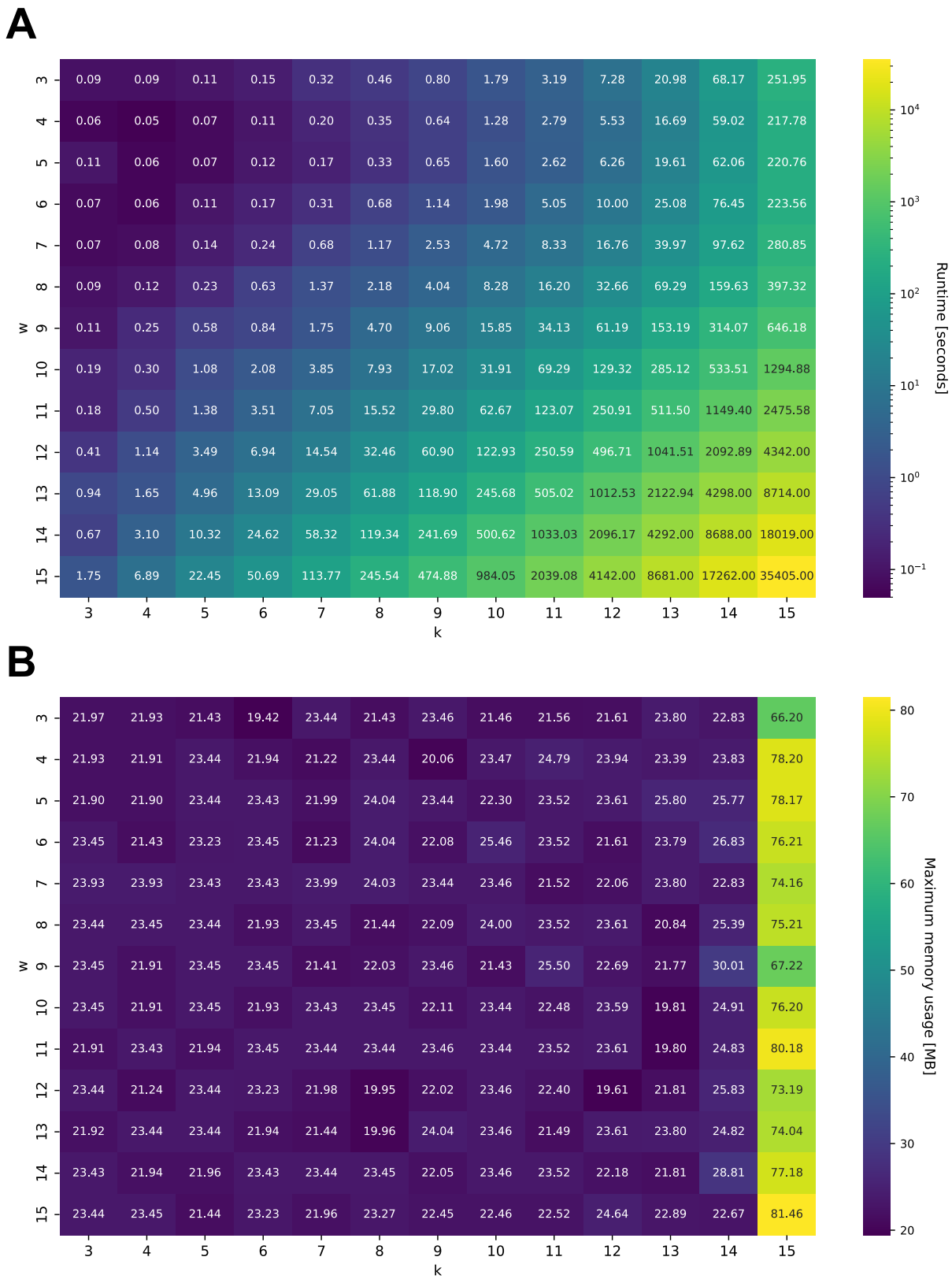

Fig. S3. Runtime (A) and maximum memory usage (B) of GM-expected on various w and k values.

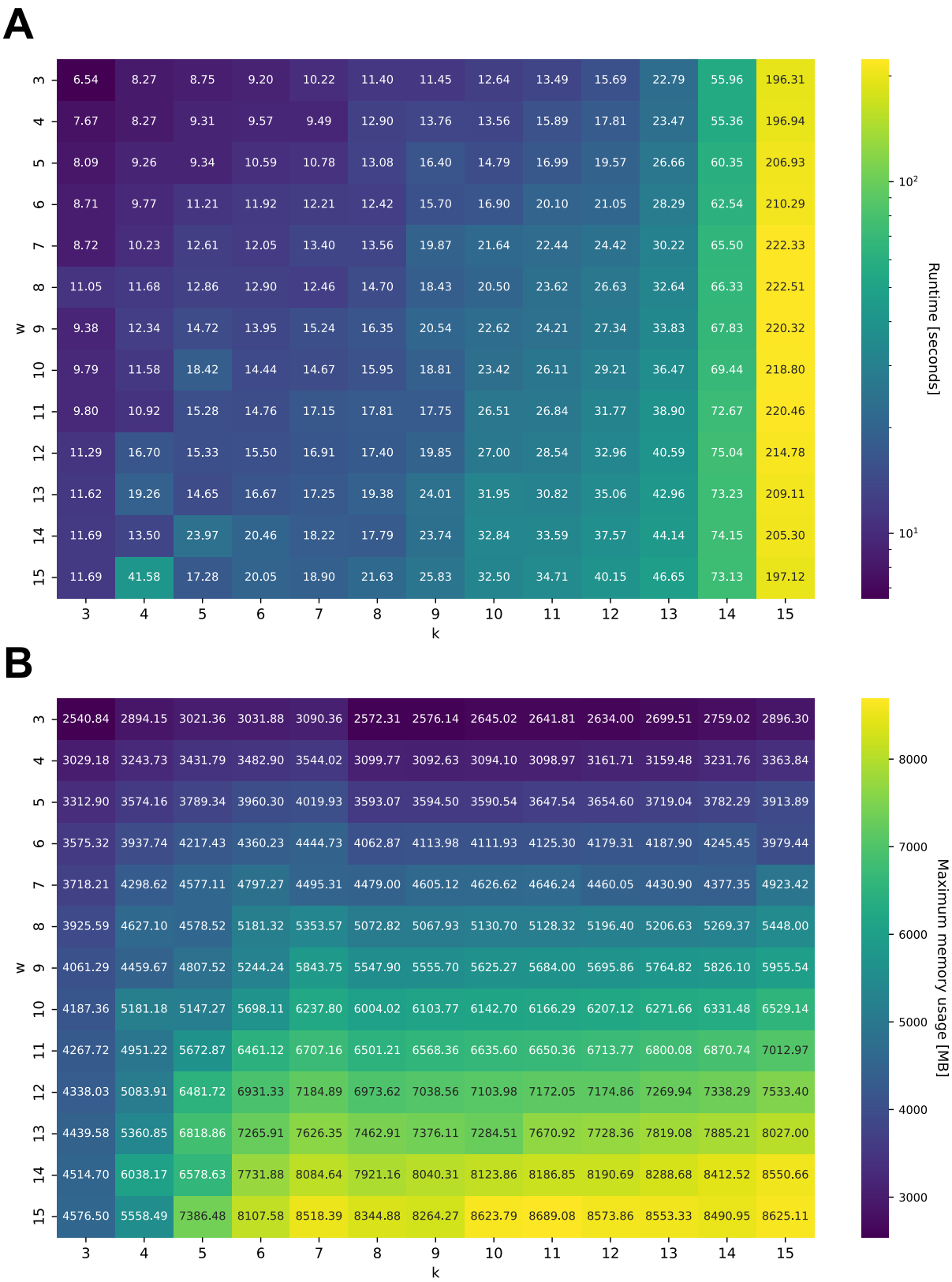

Fig. S4. Runtime (A) and maximum memory usage (B) of GM-particular on various w and k values over 1 million nucleotides from Chromosome X.