# Generating low-density minimizers

Shay Golan[1,2,†][0000−0001−8357−2802], Ido Tziony[3,†][0009−0000−5738−0882], Matan Kraus[3][0000−0002−2989−1113], Yaron Orenstein[3,*][0000−0002−3583−3112], and Arseny Shur[3,*][0000−0002−7812−3399]

[1] Reichman University, Herzliya 4610101, Israel
[2] University of Haifa, Haifa 3498838, Israel
[3] Bar-Ilan University, Ramat Gan 5290002, Israel
shur@datalab.cs.biu.ac.il, yaron.orenstein@biu.ac.il

**Abstract.** Minimizers is the most popular $k$-mer selection scheme. It is used in many algorithms and data structures analyzing high-throughput sequencing 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 smallest number of selected $k$-mers over a random DNA sequence, termed the expected density, are desired for improved performance of high-throughput sequencing analyses. Yet, no method to date exists to generate minimizers that achieve minimum expected density. Moreover, existing selection schemes fail to achieve low density for values of $k$ and $w$ that are most practical for high-throughput sequencing algorithms and data structures. Here, we present GreedyMini, a novel greedy algorithm to generate minimizers with low expected density. Moreover, we present innovative techniques to transform minimizers from binary to larger alphabets and to larger $k$ values, an extension of GreedyMini to generate minimizers that achieve low density for a particular DNA sequence, and efficient methods to calculate the exact expected density. We combine these innovations into GreedyMini+, a novel method to generate DNA minimizers for practical values of $k$ and $w$. We demonstrate over various combinations of practical $k$ and $w$ values that GreedyMini+ generates minimizers that achieve expected densities very close to a recent theoretical lower bound, and both expected and particular densities much lower compared to existing selection schemes. We expect GreedyMini+ to improve the performance of many high-throughput sequencing algorithms and data structures and advance the research of $k$-mer selection schemes.

**Keywords:** Minimizers · Minimizer orders · Forward local selection schemes · High-throughput sequencing · Universal hitting sets · $k$-mers.

† These authors contributed equally to this work.
* Shared correspondence.

## 1 Introduction

Minimizers are used in many bioinformatic applications, including sequence alignment, genome assembly, and data compression [11, 22, 12]. Minimizers are schemes that select representative sets of $k$-mers from DNA 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 [22]. Low density indicates a sparse and efficient selection, which enhances the performance of high-throughput sequencing 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 [16, 10]. 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 $k \leq 15$ [13, 14, 4]. Decycling-based minimizers can be applied to any $k$ value and achieve the lowest expected density compared to existing selection schemes [15]. Frequency-based orders can be used as a simple technique to generate minimizers with low particular density [1]. More recently, DeepMinimizer was developed to utilize machine learning to distribute selected $k$-mers more evenly, directly targeting low particular density [6]. 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 densities achieved by existing selection schemes [7]. Several other $k$-mer selection schemes (e.g., Miniception [19], mod-minimizer [5], strobemers [17], and syncmers [3]) make use of minimizers and can benefit from the use of low-density minimizers.

Here, we present GreedyMini, a novel greedy algorithm to generate minimizers with low expected density. Moreover, we present an innovative technique to transform minimizers from binary to larger alphabets, a way to extend a minimizers scheme to larger values of $k$, an extension to generate minimizers that achieve low density for a particular DNA sequence, and efficient methods to calculate the exact expected density. We prove density guarantees for our extensions to larger alphabets, and larger $k$. We combine these innovations into GreedyMini+, a novel method to generate low-density DNA minimizers. We demonstrate over various combinations of $k$ and $w$ values, that are the most practical for high-throughput sequencing analyses, that GreedyMini+ generates minimizers that achieve lower density compared to existing selection schemes and very close to a recent theoretical lower bound (e.g., within 1% for $w = k - 1$). We expect GreedyMini+ to improve the runtime and memory usage of high-throughput sequencing 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 $\Sigma$. $S[i]$ 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 an integer $f(\sigma, k, w)$ appears as the time or space complexity of an algorithm, it is assumed to fit into $O(1)$ machine words. For a boolean expression $B$, $[B]$ equals 1 if $B$ is true and 0 otherwise.

*Local selection schemes and their density.* A *local (selection) scheme* with parameters $(\sigma, k, w)$ is a map $\mathcal{L} : \Sigma^{w+k-1} \to [0, w)$. This map acts on every string $S \in \Sigma^+$, selecting one position in each $(w+k-1)$-window so that in a window $v = S[i, i+w+k-1)$ the position $i + \mathcal{L}(v)$ in $S$ is selected. The most popular metric to evaluate a local scheme by is the density of selected positions.

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

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

*Charged windows.* For a given minimizer $\mathcal{L} = (\rho, w)$, a $(w+k)$-window $v$ is *charged* if its minimum-rank $k$-mer is either its prefix or its unique suffix; otherwise, it is *free*. An important observation is that every string $S$ contains exactly $|\mathcal{L}(S)| - 1$ (not necessarily distinct) charged $(w+k)$-windows [10]. Since all possible $n$-strings have, in total, the same number of occurrences of each $(w+k)$-window, the expected density $d_{\mathcal{L}}$ of a minimizer equals the fraction of charged windows in $\Sigma^{w+k}$ [10, 19].

# 3 Methods

## 3.1 Efficiently calculating the exact expected density of a minimizer

We present two new algorithms to calculate the *exact* expected density of a given minimizer order by counting charged $(w+k)$-windows: a small-space recursive algorithm (Theorem 1) and a dynamic-programming algorithm for large $w$ (Theorem 2). Proofs of these theorems are presented in Sections A.1 and A.2.

**Theorem 1.** *If $\rho$ is a UHS order on $\Sigma^k$ for $w$, then the expected density of the minimizer $(\rho, w)$ can be computed in $O(\sigma^{k+w})$ time and $O(w)$ space.*

**Theorem 2.** *If $\rho$ is a UHS order on $\Sigma^k$ for all $w \in [w_{\min}, w_{\max} + 1)$, then the expected density of all minimizers $(\rho, w)$ with $w_{\min} \leq w \leq w_{\max}$ can be computed simultaneously in $O(w_{\max}\sigma^{2k+1})$ time and $O(\sigma^{2k})$ space.*

*Remark 1.* If computing the exact density is not feasible, it can be estimated by sampling a sequence of random $(w+k)$-windows, which is a Bernoulli process.

## 3.2 GreedyMini algorithm to generate low-density minimizers greedily

We present GreedyMini (Algorithm 1), a novel algorithm to generate low-density minimizers. Given the parameters $(\sigma, k, w)$, GreedyMini generates a UHS order $\rho$ on $\Sigma^k$ 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. The algorithm maintains a score function for all unranked $k$-mers. At each iteration it chooses uniformly at random one of the $k$-mers with the lowest score, assigns the next rank to it, and updates the scores of all other unranked $k$-mers.

Let us define the score function. 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 the greedy rule of selecting a $k$-mer the lowest score 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. The following theorem shows that GreedyMini has a low-space implementation. We prove the theorem and give the pseudocode of efficient implementations of non-trivial auxiliary functions in Section A.3.

**Theorem 3.** $\mathsf{GreedyMini}(\sigma, k, w)$ *can be implemented to run in $O(\sigma^{2k} + w\sigma^{k+w})$ time and $O(\sigma^k + w)$ space.*

**Algorithm 1** : GreedyMini

*Input:* alphabet size $\sigma$, $k$-mer length $k$, window size $w$
*Output:* UHS order $\rho$ for $w$ on $\Sigma^k$

```
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                        ▷ current rank to be assigned
    u ← select_candidate(X, Y)              ▷ selected k-mer u has the lowest |Y_u|/|X_u|
    if u = null                             ▷ no live windows, X = [0, ..., 0]
        return rank                         ▷ ρ is defined by the ranks of k-mers
    X, Y ← update_counts(X, Y, u, rank)     ▷ for the next iteration
    rank[u] ← r                             ▷ assign rank r to the selected k-mer u
```

### 3.3 Extensions of GreedyMini

*Approximate greedy.* At each iteration, GreedyMini$(\sigma, k, w)$ draws a random $k$-mer from the pool of all lowest-scored $k$-mers. We increase this pool by admitting $k$-mers that have almost lowest scores. We introduced a parameter that defines the admittance threshold of the score of $k$-mers being considered for selection at each iteration. This inclusion of additional $k$-mers 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.

*Particular greedy.* We extended GreedyMini$(\sigma, k, w)$ to generate minimizers with low particular density with respect to a specific input string $S$. In this case, each window has a weight equal to the number of its occurrences in $S$. We compute and store 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 is bounded by the observed windows in $S$ and by $\sigma^{w+k}$. Then, the only difference with the original GreedyMini is that the arrays X and Y contain *sums of weights* of windows rather than the numbers of windows. More efficient implementations of this extension of GreedyMini exist in the case where $|S| \ll \sigma^{w+k}$ by using a linked list of non-zero windows.

### 3.4 Hill-climbing optimization of minimizer orders

We apply a randomized version of the standard local search technique called *hill climbing* to improve a given minimizer $(\rho, w)$, where $\rho$ is a UHS order for $w$. The function Swap$(\rho, w)$, performing one iteration of the search, selects a random rank $i$ such that both $i$ and $i{+}1$ are assigned in $\rho$ and defines a UHS order $\rho'$ obtained from $\rho$ by swapping the $k$-mers with ranks $i$ and $i{+}1$. Swap calculates $\Delta_{\rho,\rho'} \triangleq d_{(\rho,w)} - d_{(\rho',w)}$. If $\Delta_{\rho,\rho'} > 0$, Swap replaces $\rho$ with $\rho'$; if $\Delta_{\rho,\rho'} < 0$, Swap does nothing; if $\Delta_{\rho,\rho'} = 0$, Swap replaces $\rho$ with $\rho'$ with probability 0.5. The search stops after a prescribed number of iterations rather than at a local minimum since we observed that most $\Delta_{\rho,\rho'}$ are zero, and since we developed a fast implementation of Swap (Theorem 4), which allows for many iterations. The proof of Theorem 4 and the pseudocode of Swap are presented in Section A.4.

**Theorem 4.** *Let $\rho$ be a UHS order on $\Sigma^k$ for $w$. The function $\mathsf{Swap}(\rho, w)$ can be implemented in $O(\sigma^k + \sigma^w)$ time and $O(w)$ additional space.*

### 3.5 Transformation to larger alphabets, and to larger $k$

We present novel approaches to transform any minimizer to a minimizer over a larger alphabet or with increased $k$ with provable density guarantees. We define *dual minimizers* as local schemes that act like minimizers but break ties to the right. For a minimizer $\mathcal{L}$, we denote the dual minimizer using the same order as $\mathcal{L}$ by $\mathcal{L}^{\leftarrow}$. Notice that $d_{\mathcal{L}}$ and $d_{\mathcal{L}^{\leftarrow}}$ 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 $\rho$ for $w$ on $\Sigma^k$, 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.

**Lemma 1.** *If $\rho$ 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 $\rho$-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 define a transformation of an arbitrary minimizer $\mathcal{L}$ with the parameters $(\sigma, k, w)$ to a minimizer $\mathcal{L}'$ with the parameters $(\sigma\gamma, k, w)$ for integer $\gamma > 1$, and prove a density guarantee for $\mathcal{L}'$. 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 *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 $\rho$ for $w$ on $\Sigma^k$ and a linear order $\tau$ on $\Gamma^k$, 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 $(\rho^{\leftarrow} \times \tau, w)$.

**Theorem 5.** *Let $\rho$ be a UHS order on $\Sigma^k$ for $w$ and $\tau$ be a linear order on $\Gamma^k$. Then, $\min\{d_{(\rho\times\tau, w)}, d_{(\rho^{\leftarrow}\times\tau, w)}\} \leq (d_{(\rho,w)} + d_{(\rho,w)^{\leftarrow}})/2$.*

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

*Claim.* $|B| \leq |G^{\leftarrow}|$.

*Proof (of the Claim).* Let $v \in B$ and let $u$ be the minimum-rank $k$-mer of $v$. Then, $u_\Sigma$ has the minimum $\rho$-rank among the $k$-mers of $v_\Sigma$. By the definition of bad, $v_\Sigma$ is not charged by $\rho$, so $u_\Sigma$ is neither a prefix nor a unique suffix of $v_\Sigma$. Hence, $u$ is a suffix of $v$, and $u_\Sigma$ occurs more than once in $v_\Sigma$. 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 $u_\Sigma$ in $v_\Sigma$ (Figure 1).

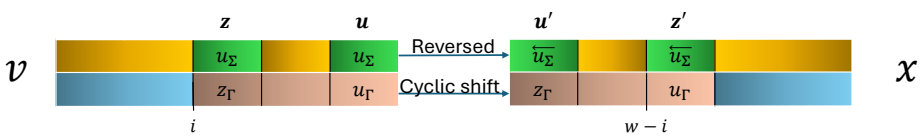


Fig. 1: Mapping a bad (for $\rho \times \tau$) window $v$ to a good (for $\rho^{\leftarrow} \times \tau$) window $x$.

By the definition of mirror, the prefix $\overleftarrow{u_\Sigma}$ of $x_\Sigma$ has the minimum $\rho^{\leftarrow}$-rank among the $k$-mers of $x_\Sigma$. Then, $x_\Sigma$ 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 $u_\Sigma$ is not a prefix of $v_\Sigma$. As $\overleftarrow{u_\Sigma}$ has the minimum $\rho^{\leftarrow}$-rank among the $k$-mers of $x_\Sigma$, 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, $\phi$ maps $B$ to $G^{\leftarrow}$.

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 $x_\Gamma, x'_\Gamma$ are obtained by the same shift of $v_\Gamma$ and $v'_\Gamma$, 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 $\phi$ is injective. The claim now follows. □

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

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

The result now follows by Lemma 1. □

We use a similar approach to transform a minimizer with parameters $(\sigma, k, w)$ to a minimizer with the parameters $(\sigma, k{+}1, w)$. Given a UHS order $\rho$ on $\Sigma^k$ for $w$, we define its *extensions* $\rho_1$ and $\rho_2$ on $\Sigma^{k+1}$ 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 $\mathsf{rank}_{\rho_2}(au) = \sigma{\cdot}\mathsf{rank}_{\rho^{\leftarrow}}(u) + \sigma - a - 1$. (A simplified version of this idea was used in [20], but their resulting scheme is not a minimizer.) We prove the next theorem in Section A.5.

**Theorem 6.** *Let $\rho$ be a UHS order on $\Sigma^k$ for $w$, and let $\rho_1, \rho_2$ be its extensions. Then,* $\min(d_{(\rho_1,w)}, d_{(\rho_2,w)}) \leq (d_{(\rho,w)} + d_{(\rho,w)^{\leftarrow}})/2$.

### 3.6 GreedyMini+ method to generate low-density DNA minimizers

We combined the techniques from Sections 3.1–3.5 into GreedyMini+, a novel method to generate low-density DNA minimizers for given parameters $k$ and $w$. First, we generate a low-density binary minimizer $(\rho, w)$ as described below. Then, we apply Theorem 5 for $\sigma = \gamma = 2$ and output the DNA minimizer achieving lower density among $(\rho \times \tau, w)$ and $(\rho^{\leftarrow} \times \tau, w)$, where $\tau$ is lexicographic. The output minimizer has low density and requires only $O(2^k)$ space.

Generating a low-density binary minimizer includes three consecutive stages. *Stage 1*: We run multiple instances (by default 4096) of GreedyMini$(2, k, w)$ and select the minimizer charging the minimum number of windows or its mirror if it charges less windows. We use the approximate GreedyMini (Section 3.3) by considering each unranked $k$-mer $u$, whose *inverted score* $\frac{\mathsf{X}[u]-\mathsf{Y}[u]}{\mathsf{Y}[u]}$ is at least $\alpha I$, where $I$ is the maximum inverted score and $\alpha \in [0.94..0.9996]$ is a parameter. $1-\alpha$ is drawn from a log-uniform distribution for each run of GreedyMini.
*Stage 2*: For the selected minimizer, we run an instance of the local search (Section 3.4) on each available CPU core (64 on our machine) with the runtime bounded by the runtime of Stage 1 (in practice, $>400 \cdot 2^k$ Swap calls per core). We select the minimizer with the smallest number of charged windows.
*Stage 3 (included in our tests over consecutive $k$ values)*: We compare $d(\rho, w)$ to $d(\rho', w)$, where $(\rho', w)$ is the minimizer generated for $(2, k-1, w)$. If $d(\rho, w) > d(\rho', w)$, we extend $\rho'$ to a UHS order of $k$-mers using Theorem 6 and run Stage 2 over the obtained minimizer.

*Remark 2.* If $w$ is prohibitively large for GreedyMini, we run GreedyMini+ for $(k, w')$ with some $w' < w$. In Section 4, we show empirically that as long as $w' > k$, the resulting minimizer is still competitive for the window size $w$.

To construct minimizers with low *particular* density for the DNA alphabet, we extended GreedyMini to work with weighted windows (Section 3.3) and we run only Stage 1 as GreedyMiniParticular+.

## 4 Results

We used GreedyMini+ to generate low-density DNA minimizers for various $k, w$ combinations. First, we ran GreedyMini+ over all $3 \leq k, w \leq 15$. Second, we ran GreedyMini+ with fixed $w \in \{8, 12\}$ and $3 \leq k \leq 38 - w$ (we extend for $k > 17$ and $k > 16$, respectively, using Theorem 6) and with fixed $k \in \{8, 12\}$ and $3 \leq w \leq 100$ (we extend for $w > 19$ and $w > 16$, respectively, using Remark 2). Prior to the extension point, we computed the expected density of resulting minimizers exactly if $k + w < 27$, while for $27 \leq k + w \leq 30$, we sampled $10^8 \cdot 2^{w+k}$ $(w + k)$-windows (all windows over $\Sigma$ times $10^8$ windows over $\Gamma$, where $\sigma = \gamma = 2$). After the extension point, we report the upper bound of Theorem 5, computing the expected density of the binary order exactly (Theorems 1 and 2). We compared the densities achieved by GreedyMini+ minimizers

to densities achieved by state-of-the-art selection schemes as implemented in the mod-minimizer GitHub repository using a random 4-ary $10^7$-string [8].

We also ran GreedyMiniParticular+ to generate minimizers with low particular density for a DNA sequence of $10^6$ nucleotides from Chromosome X for $w = 12$ and $3 \leq k \leq 16$, and for $k = 8$ and $3 \leq w \leq 16$. We compared the results to the same set of selection schemes; our attempts to compare to DeepMinimizer [6] and polar sets [21] failed, because their GitHub implementations raised errors and lacked documentation (Section A.6). We report the maximum runtime and memory usage of GreedyMini+ and GreedyMiniParticular+ in Section A.7.

For fixed $w$ (Figures 2A,B), GreedyMini+ outperformed all existing selection schemes over most $k$ values, losing to mod-minimizers or syncmers only in the impractical range where $k$ is significantly larger than $w$. For fixed $k$ (Figures 2C,D), GreedyMini+ also outperformed existing schemes, with double-decycling closely behind for $k = 12$. We observed a similar trend for particular density (Figures 2E,F), where the best performers were GreedyMiniParticular+ and GreedyMini+, with GreedyMiniParticular+ achieving clearer superiority as $w + k$ increases (e.g., for $w + k > 24$).

At $w \in \{k, k-1\}$ for expected density (Figures 2G,H), GreedyMini+ generated minimizers that are within 1% of the theoretical lower bound for forward local schemes [7], which constitute a wide class of schemes including minimizers. Inspired by such a small gap, we note that the binary minimizers generated by GreedyMini$(2, k, w)$ hit the lower bound for the following $(k, w)$ pairs: $(4, 3)$, $(7, 3)$, $(5, 4)$, $(10, 3)$ and $(6, 5)$, with the latter two requiring more runs with a smaller $\alpha$. Thus, we conjecture that within the range where GreedyMini+ is feasible to run, the best orders obtained by GreedyMini+ have densities very close to the lower bound. Based on this conjecture, we present in Figure 3 the landscape of densities 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 novel method to generate low-density DNA minimizers for given $k$, and $w$, by using the GreedyMini algorithm to generate binary minimizers and transforming them to the DNA alphabet and larger $k$ if needed. We evaluated the performance of GreedyMini+ by calculating the achieved expected and particular densities, and we conclude that it generates minimizers that achieve the lowest density compared to existing selection schemes over practical values of $k$ and $w$ (e.g., the very popular Minimap2 [9], Kraken [18], and KMC2 [2] use $(k, w)$ pairs of $(15, 10)$, $(15, 17)$, and $(7, 49)$, respectively). 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 local forward schemes for the binary alphabet and several $(k, w)$ pairs.

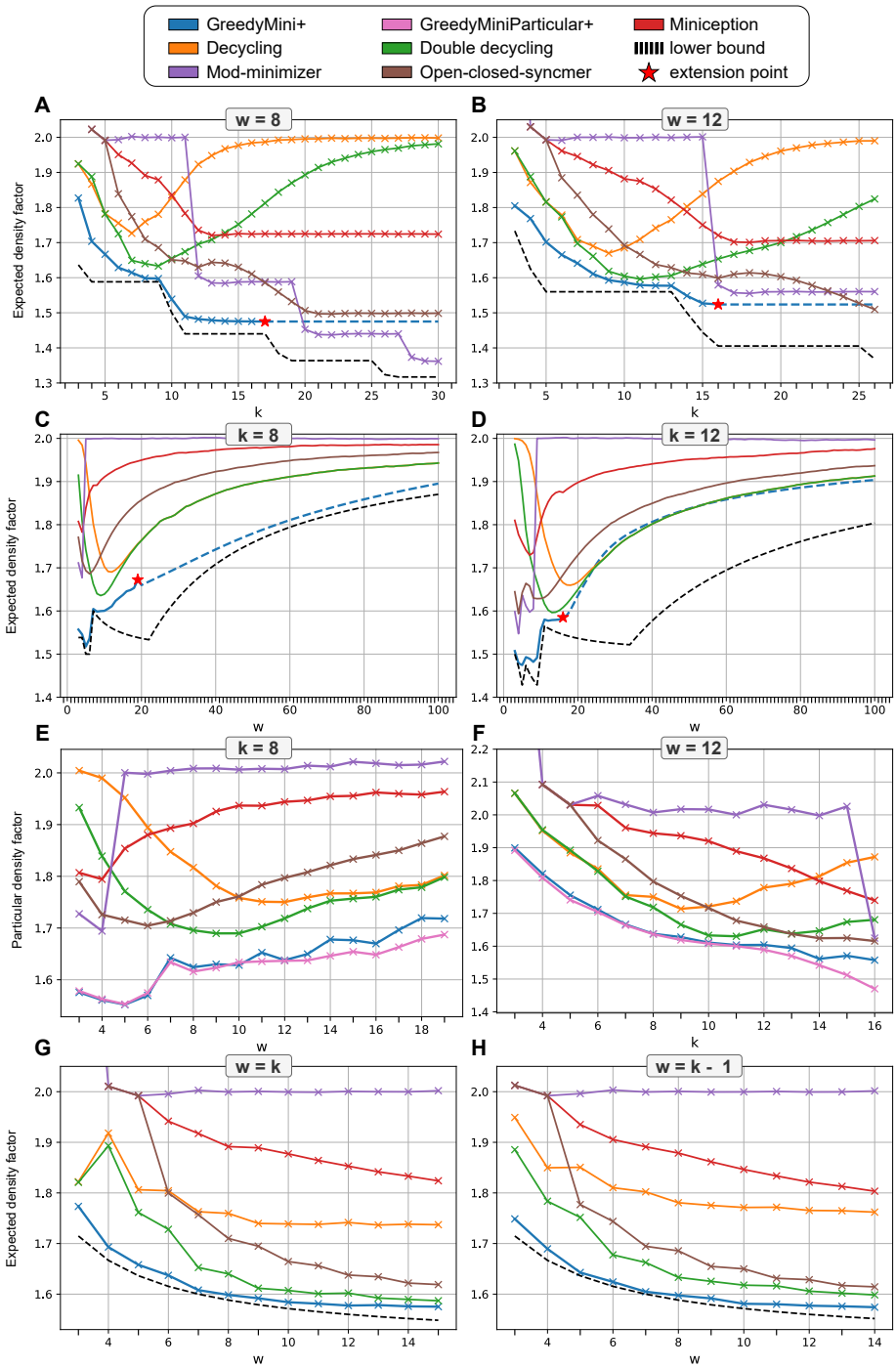


Fig. 2: Performance of various DNA $k$-mer selection schemes over various $(k, w)$ values. (A-D) Expected density comparisons over fixed $k$ or $w$. (E,F) Particular density comparisons over 1 million nucleotides from Chromosome X. (G,H) Expected density comparisons over $w \in \{k, k-1\}$.

The main limitation of GreedyMini+ is the exponential dependence in runtime on $w + k$. But thanks to our novel transformation of minimizers to larger alphabets (Theorem 5), this dependence is only $O(w2^{w+k})$ per run of GreedyMini. This is an acceptable runtime for a wide range of practical values of $k$ and $w$. This range is further extended to larger $k$ values due to Theorem 6 and to larger $w$ values in view of Remark 2. Similarly, Theorem 5 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. For larger $k$, one can generate a minimizer for $k = 20$ and extend it multiple times, such that the lookup table occupies only 4MB.

Our study raises several open questions. First, for which $(\sigma, k, w)$ combinations minimizers are optimal among all forward local 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 GreedyMini, 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 GreedyMini without storing the order explicitly?

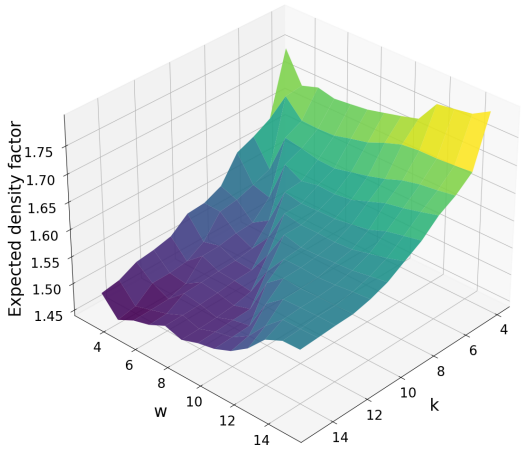


Fig. 3: Density factors of the binary minimizers generated as part of GreedyMini+ testing.

There are several directions for future work. First, we plan to plug our new minimizers in the Miniception, mod-minimizer, open-and-closed syncmers, and strobemers schemes to improve their density, and study their effect 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 algorithms and data structures and observe their improved performance in runtime and memory usage.

**Code availability:** https://github.com/OrensteinLab/GreedyMini

**Acknowledgment:** Ido Tziony acknowledges the cloud-computing credit support of the Israel Data Science and AI Initiative.

**Funding:** Shay Golan is supported by Israel Science Foundation grant no. 810/21. 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. R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev, *On the representation of de Bruijn graphs*, Journal of Computational Biology, 22 (2015), pp. 336–352.
2. S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, *KMC 2: fast and resource-frugal k-mer counting*, Bioinformatics, 31 (2015), pp. 1569–1576.
3. R. Edgar, *Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences*, PeerJ, 9 (2021), p. e10805.
4. B. Ekim, B. Berger, and Y. Orenstein, *A randomized parallel algorithm for efficiently finding near-optimal universal hitting sets*, in International Conference on Research in Computational Molecular Biology, Springer, 2020, pp. 37–53.
5. R. Groot Koerkamp and G. E. Pibiri, *The mod-minimizer: A simple and efficient sampling algorithm for long k-mers*, in 24th International Workshop on Algorithms in Bioinformatics (WABI 2024), vol. 312, Springer, 2024, pp. 11:1–11:23.
6. M. Hoang, H. Zheng, and C. Kingsford, *Differentiable learning of sequence-specific minimizer schemes with DeepMinimizer*, Journal of Computational Biology, 29 (2022), pp. 1288–1304.
7. B. Kille, R. Groot Koerkamp, D. McAdams, A. Liu, and T. Treangen, *A near-tight lower bound on the density of forward sampling schemes*, bioRxiv, (2024), pp. 2024–09.
8. R. G. Koerkamp, *minimizers - a minimizer scheme implementation in Rust*. https://github.com/RagnarGrootKoerkamp/minimizers, 2023. Accessed: 2024-10-04.
9. H. Li, *Minimap2: pairwise alignment for nucleotide sequences*, Bioinformatics, 34 (2018), pp. 3094–3100.
10. G. Marçais, D. Pellow, D. Bork, Y. Orenstein, R. Shamir, and C. Kingsford, *Improving the performance of minimizers and winnowing schemes*, Bioinformatics, 33 (2017), pp. i110–i117.
11. C. Moeckel, M. Mareboina, M. A. Konnaris, C. S. Chan, I. Mouratidis, A. Montgomery, N. Chantzi, G. A. Pavlopoulos, and I. Georgakopoulos-Soares, *A survey of k-mer methods and applications in bioinformatics*, Computational and Structural Biotechnology Journal, 23 (2024), pp. 2289–2303.
12. M. Ndiaye, S. Prieto-Baños, L. M. Fitzgerald, A. Yazdizadeh Kharrazi, S. Oreshkov, C. Dessimoz, F. J. Sedlazeck, N. Glover, and S. Majidian, *When less is more: sketching with minimizers in genomics*, Genome Biology, 25 (2024), p. 270.
13. Y. Orenstein, D. Pellow, G. Marçais, R. Shamir, and C. Kingsford, *Compact universal k-mer hitting sets*, in Algorithms in Bioinformatics: 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22-24, 2016. Proceedings 16, Springer, 2016, pp. 257–268.
14. Y. Orenstein, D. Pellow, G. Marçais, R. Shamir, and C. Kingsford, *Designing small universal k-mer hitting sets for improved analysis of high-throughput sequencing*, PLoS Computational Biology, 13 (2017), p. e1005777.
15. D. Pellow, L. Pu, B. Ekim, L. Kotlar, B. Berger, R. Shamir, and Y. Orenstein, *Efficient minimizer orders for large values of k using minimum decycling sets*, Genome Research, 33 (2023), pp. 1154–1161.

16. M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, *Reducing storage requirements for biological sequence comparison*, Bioinformatics, 20 (2004), pp. 3363–3369.
17. K. Sahlin, *Effective sequence similarity detection with strobemers*, Genome Research, 31 (2021), pp. 2080–2094.
18. D. E. Wood and S. L. Salzberg, *Kraken: ultrafast metagenomic sequence classification using exact alignments*, Genome Biology, 15 (2014), pp. 1–12.
19. H. Zheng, C. Kingsford, and G. Marçais, *Improved design and analysis of practical minimizers*, Bioinformatics, 36 (2020), pp. i119–i127.
20. H. Zheng, C. Kingsford, and G. Marçais, *Lower density selection schemes via small universal hitting sets with short remaining path length*, Journal of Computational Biology, 28 (2021), pp. 395–409.
21. H. Zheng, C. Kingsford, and G. Marçais, *Sequence-specific minimizers via polar sets*, Bioinformatics, 37 (2021), pp. i187–i195.
22. H. Zheng, G. Marçais, and C. Kingsford, *Creating and using minimizer sketches in computational genomics*, Journal of Computational Biology, 30 (2023), pp. 1251–1276.

# A Appendix

*Prefix trees.* Given a set $U$ of strings, we define its *prefix tree* (or *trie*) $\mathcal{T}_U$. 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 $a \in \Sigma$. We use prefix trees of sets of windows, computing node characteristics during a depth-first search (DFS) without explicitly storing the whole tree. We assume that each node $z$ with $|z| \geq k$ is labeled by its $k$-suffix.

*Strings as integers.* In our algorithms, we view $k$-mers and explicitly stored windows as **integers written in $\sigma$-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 3 and 4, the integer operations are made explicit; the shift operation $v \gg_\sigma n$ deletes the last $n$ digits (characters) of $v$.

## A.1 Proof of Theorem 1

To calculate the density $d_{(\rho,w)}$, we count the $(w+k)$-windows charged by $\rho$ and divide the result by $\sigma^{w+k}$. We focus on counting charged windows. Let $U$ be the set of all windows with some $k$-prefix $u$. The prefix tree $\mathcal{T}_U$ of $U$ consists of the path from the root to $u$ and a complete $\sigma$-ary tree of depth $w$, rooted at $u$. In order to count charged windows with the prefix $u$, we run a recursive DFS on this complete subtree by calling $\mathsf{DFS}(u, u, 0, \infty)$ (Algorithm 2).

With each node $z$ we associate the minimum rank $r_z$ of a $k$-mer occurring in $z$, i.e., $r_z = \min\{\mathsf{rank}_\rho(u') \mid u' \text{ occurs in } z\}$. Let $z'$ be the label of $z$. If $z$ is internal, it gets the minimum rank $r$ of its parent and computes $r_z = \min\{r, \mathsf{rank}(z')\}$. Then, it recursively calls all its children. Finally, $z$ returns the number of charged windows in its subtree, summing up the values returned from its children. Let $z$ be a leaf, i.e., a window in $U$. It returns 1 if it is charged and 0 otherwise. Let $r$ be the rank of the parent of $z$; then $z$ returns 1 if $\mathsf{rank}(u) = r$ or $\mathsf{rank}(z') < r$.

**Algorithm 2** : function DFS for recursive count of charged $(w+k)$-windows

*Input:* $k$-prefix $u$, current node $z$, depth $d$, parent's minimum rank $r$; global $\sigma, k, w$
*Output:* number of windows with the prefix $z$, charged by $\rho$

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

The above algorithm spends $O(\sigma)$ time per internal node 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. Running it for every $k$-mer $u$, we compute the number of windows charged by $\rho$ within the required complexity bounds. □

*Remark 3.* Since $\rho$ is a UHS order, it is sufficient to call DFS only for sets of windows with a *ranked* $k$-prefix and then a symmetric (up to minor adjustments) function for the sets of windows having *ranked* $k$-suffixes. If the number of ranked $k$-mers in $\rho$ is close to the size of a minimum decycling set, which is $\approx \sigma^k/k$ and a lower bound for any UHS, this approach significantly speeds up the computation.

### A.2 Proof of Theorem 2

We count charged windows using a dynamic-programming algorithm, making use of the tables $\mathsf{pref}[r,u,w]$, $\mathsf{int}[r,u,w]$, and $\mathsf{suff}[u,w]$ to store the numbers of $(w{+}k)$-windows with the $k$-suffix $u$ and the following additional properties:

- $\mathsf{pref}[r,u,w]$: the minimum rank of a $k$-mer is $r$, attained by the $k$-prefix;
- $\mathsf{int}[r,u,w]$: the minimum rank of a $k$-mer is $r$, attained by some internal $k$-mer and *not* by the $k$-prefix;
- $\mathsf{suff}[u,w]$: the $k$-suffix $u$ is the unique $k$-mer of the minimum rank.

If $\rho$ is a UHS order for $w$, then the minimum rank of a $k$-mer in every window is finite. Hence, if $\rho$ has $m$ ranked $k$-mers, the number of $(w{+}k)$-windows charged by $\rho$ is $\sum_{u\in\Sigma^k}\big(\mathsf{suff}[u,w]+\sum_{r=0}^{m-1}\mathsf{pref}(r,u,w)\big)$.

If $x\in\Sigma^{k-1}$, $a,b\in\Sigma$, then we call the $k$-mer $bx$ *previous* to the $k$-mer $xa$. We denote the previous $k$-mers of a $k$-mer $u$ by $u_1,\ldots,u_\sigma$. The base case for the dynamic-programming rules is $w=0$. From definitions of the tables $\mathsf{pref}[r,u,0]=1$ if $\mathsf{rank}[u]=r$, while all other base-case entries of all tables are zeroes. The recursive rules are straightforward from the definitions:

$$\mathsf{pref}[r,u,w{+}1]=\sum_{i=1}^{\sigma}[\mathsf{rank}_\rho(u)\ge r]\cdot\mathsf{pref}[r,u_i,w]$$

$$\mathsf{int}[r,u,w{+}1]=\sum_{i=1}^{\sigma}\Big([\mathsf{rank}_\rho(u)\ge r]\cdot\mathsf{int}[r,u_i,w] + [\mathsf{rank}_\rho(u)\ge\mathsf{rank}_\rho(u_i)=r]\cdot\mathsf{suff}[u_i,w]\Big)$$

$$\mathsf{suff}[u,w{+}1]=\sum_{i=1}^{\sigma}\Bigg(\sum_{r>\mathsf{rank}_\rho(u)}(\mathsf{pref}[r,u_i,w]+\mathsf{int}[r,u_i,w]) + [\mathsf{rank}_\rho(u)<\mathsf{rank}_\rho(u_i)]\cdot\mathsf{suff}(u_i,w)\Bigg)$$

Computing the tables for $w+1$ takes $O(\sigma)$ time per each of $O(\sigma^{2k})$ entries of pref and int, and $O(\sigma^{k+1})$ time per each of $O(\sigma^k)$ entries of suff, which is $O(\sigma^{2k+1})$ in total. As it suffices to store in memory only two layers of the tables (i.e., those for $w$ and $w+1$), we obtain the claimed time and space bounds. □

*Remark 4.* The actual size of each of the tables pref and int is $(m+1)\sigma^k$, because the rank in $\rho$ takes $m$ finite and one infinite value. So if $\rho$ is close to a minimum decycling set, the tables are smaller and can be computed faster.

### A.3 Proof of Theorem 3

We maintain arrays $\mathsf{X}, \mathsf{Y}$, and rank, each of size $\sigma^k$. 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 $\mathsf{X}$ and $\mathsf{Y}$ 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 $\sigma^k$ iterations follow, each consisting of two steps: choosing the $k$-mer to assign rank to and updating the arrays $\mathsf{X}, \mathsf{Y}$ (for convenience, the rank is assigned *after* the updates). Computing the lowest score and choosing a $k$-mer with this score can be done in time proportional to the size of $\mathsf{X}$ and $\mathsf{Y}$.

Assume that the $k$-mer $u$ is chosen and consider the update step (Algorithm 3). For every $i \in [0, w+1)$, exactly $\sigma^w$ 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 $\sigma^i$. Hence, the total number of $u$-blocks is at most $\sum_{i=0}^{w} \sigma^i = O(\sigma^w)$.

**Algorithm 3** : function update_counts

*Input:* Arrays $\mathsf{X}$, $\mathsf{Y}$, rank, $k$-mer $u$; global $\sigma, k, w$
*Output:* Arrays $\mathsf{X}', \mathsf{Y}'$ updated from $\mathsf{X}$ and $\mathsf{Y}$

$v, d \leftarrow \mathsf{nextblock}(0^{w+k}, u)$ ▷ first block of windows containing $u$ (at position $d$)
**while** $v < \infty$ ▷ current block exists
  $\mathsf{seen} \leftarrow \varnothing$
  **for** $i \leftarrow 0$ to $d-1$ ▷ processing $k$-mers starting to the left of $u$
    $z \leftarrow (v \gg_\sigma (w-i)) \bmod \sigma^k$ ▷ $k$-mer starting at position $i$
    **if** $\mathsf{rank}[z] < \infty$ ▷ no live windows in this block, move to next
      $v, d \leftarrow \mathsf{nextblock}((v \gg_\sigma (w-i)) + 1)) \cdot \sigma^{w-i}, u)$; **break**
    $\mathsf{seen} \leftarrow \mathsf{seen} \cup \{z\}$
  $L, \mathsf{X}', \mathsf{Y}' \leftarrow \mathsf{DFS\_count\_and\_update}(u, d, \mathsf{seen}, \mathsf{X}, \mathsf{Y})$ ▷ Algorithm 5
  **for** $z \in \mathsf{seen}$ ▷ subtract the number of live windows in the current block
    $\mathsf{X}'[z] \leftarrow \mathsf{X}'[z] - L$
  $\mathsf{Y}'[v \gg_\sigma w] \leftarrow \mathsf{Y}'[v \gg_\sigma w] - L$ ▷ for the prefix of $v$ live windows are charged
  $v, d \leftarrow \mathsf{nextblock}((v \gg_\sigma (w-d)) + 1)) \cdot \sigma^{w-i}, u)$ ▷ move to the next block
**return** $\mathsf{X}', \mathsf{Y}'$

We process one block at a time, navigating between blocks with the help of an auxiliary function $\mathsf{nextblock}(v, u)$ (Algorithm 4): it returns the smallest window $v'$ that is greater than the window $v$ and contains $u$, or $\mathsf{null}$ if no such window exists. In other words, $\mathsf{nextblock}(v, u)$ returns the first window of the next $u$-block. To implement $\mathsf{nextblock}(v, u)$, 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, $\mathsf{nextblock}(v, u)$ can be computed in $O(w)$ time.

**Algorithm 4** : function nextblock

*Input:* window $v$, $k$-mer $u$; global $\sigma, k, w$
*Output:* 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
    z ← (v mod σ^(w+k−i)) ≫_σ (w − i)           ▷ the k-mer starting at position i in m
    if z < u                                    ▷ replace u with z and the rest with 0's
        x ← ((v ≫_σ (w−i)) + u − z) · σ^(w−i)
    else if z > u                               ▷ 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
    if x < min{v', σ^(w+k)}                     ▷ smaller window found
        v' ← x; d ← i
return v', d
```

Consider processing a $u$-block $U$ (one iteration of the **while** cycle in Algorithm 3). 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 prefix tree 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 tree consists of a single path from the root to the node $pu$ followed by a complete $\sigma$-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 $\mathsf{X}$ and $\mathsf{Y}$ are zeroes). Otherwise, we put all of them in a set $\mathsf{seen}$ of occurred $k$-mers, associate $\mathsf{seen}$ with the node $pu$ of $\mathcal{T}_U$, and run a recursive DFS (Algorithm 5) 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 $z' \in \mathsf{seen}$.

**Algorithm 5** : function DFS_count_and_update

*Input:* $k$-mer $z$, depth $d$, set seen of occurred $k$-mers, arrays X, Y; global $\sigma, k, w$
*Output:* number $L$ of live windows below $z$, arrays $\mathsf{X}', Y'$ updated from $X$ and $Y$

```
if X[z] = 0                    ▷ 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                    ▷ current node is a leaf (live window)
        if f                    ▷ z is a unique suffix, decrement both X'[z], Y'[z]
            X'[z] ← X'[z] − 1; Y'[z] ← Y'[z] − 1
        return 1, X', Y'
    else                        ▷ current node is internal
        if f
            seen ← seen ∪ {z}   ▷ add z to occurred k-mers
        ℓ ← 0                   ▷ initialize counter of live windows
        for i ← 0 to σ − 1      ▷ 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
        if f                    ▷ subtract live windows from X'[z]
            X'[z] ← X'[z] − ℓ; seen ← seen \ {z}
        if d = 0                ▷ z is a prefix, decrease Y'[z]
            Y'[z] ← Y'[z] − ℓ
        return ℓ, X', Y'
```

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 $\mathsf{nextblock}$ 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 $O(2^k)$ 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 $\mathsf{seen}$ of $O(w)$ size plus $O(1)$ numbers per each of $O(w)$ active calls. Adding the size of the arrays $\mathsf{X}, \mathsf{Y}$, and $\mathsf{rank}$, we get the desired space bound. □

*Remark 5.* The number of iterations of $\mathsf{GreedyMini}(\sigma, k, w)$ equals the size of the obtained UHS, which in practice is smaller than $\sigma^k$, and in most cases just slightly larger than $\sigma^k/k$.

### A.4 Proof of Theorem 4

We focus on the efficient computation of the difference $d_{(\rho,w)} - d_{(\rho',w)}$, since all other operations are straightforward (Algorithm 6). Let $u$ and $u'$ be the swapped $k$-mers, $\mathsf{rank}[u] = i$, $\mathsf{rank}[u'] = i+1$. We do not compute $d_{(\rho',w)}$ directly, because it would take too much time. Instead, we analyse only those $(w+k)$-windows that could have different status (free/charged) for $\rho$ and $\rho'$. If a window $v$ changed its status, then its minimum-rank $k$-mer in $\rho$ is $u$ and its minimum-rank $k$-mer in $\rho'$ 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 $O(\sigma^w)$.

**Algorithm 6** : function $\mathsf{Swap}$ for swapping two $k$-mers with consecutive ranks

*Input:* array $\mathsf{rank} = \mathsf{rank}_\rho$ for a UHS order $\rho$ for $w$; global $\sigma, k, w$
*Output:* array $\mathsf{rank}$ after an attempted swap of two consecutive ranks

```
i ← random number from [0, max{rank[z] | rank[z] < ∞})
u ← ρ(i); u' ← ρ(i+1)
c ← pr_cost(rank, i, u, u', u, 0, False) + sf_cost(rank, i, u, u', u, 0, False)    ▷ Alg. 7, 8
c' ← pr_cost(rank, i, u, u', u', 0, False) + sf_cost(rank, i, u, u', u', 0, False)
if c > c'
    swap the elements rank[u] and rank[u']
else if c = c'
    swap the elements rank[u] and rank[u'] with probability 0.5
return rank
```

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 $< i$, then it has the same status in both $\rho$ and $\rho'$. Otherwise, it is charged in $\rho$; this charged window is free in $\rho'$ if it contains $u'$, but not as

a unique suffix. Respectively, it suffices to count the windows that are charged in $\rho$ and free in $\rho'$, as the opposite change of status is impossible. Similar to the proof of Theorem 3, we make use of the prefix tree $\mathcal{T}_U$ of $U$. It consists of a path from the root to $u$, followed by a complete $\sigma$-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.

During the DFS (Algorithm 7), each node returns the number of windows in its subtree that change their status. For a node $z$ with the label $z'$, if $\mathsf{rank}[z'] < i$, 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 $U$, counting the windows that are charged in $\rho'$ and free in $\rho$.

**Algorithm 7** : function pr_cost for recursive count of status-changing $k$-mers

*Input:* array rank, rank $i$, candidate $k$-mers $u, u'$ to swap, current node with $k$-suffix $z$ and depth $d$, flag $f$ (set if $u'$ is already seen); global $\sigma, k, w$

*Output:* 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] < i                                  ▷ status of windows is independent of u, u'
    return 0
if d = w                                        ▷ current node is a leaf (window)
    return [f]
else                                            ▷ current node is internal
    f ← f ∨ (z = u')
    c ← 0
    for a ∈ Σ                                   ▷ recursive call for children
        c ← c + pr_cost(rank, i, u, u', (σz) mod σ^k + a, d + 1, f)
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 $< i$ or it does not contain $u'$, then it has the same status in $\rho$ and $\rho'$. If $v$ has the prefix $u$ or $u'$, we ignore it to avoid double count. Otherwise, $v$ is free in $\rho'$ and charged in $\rho$ if and only if it has a single occurrence of $u$. We process $\bar{U}$ similar to $U$, counting the windows that are charged in $\rho$ and free in $\rho'$. The main difference is that we use the *reversed* prefix tree $\bar{\mathcal{T}}_{\bar{U}}$ of $\bar{U}$: it has *suffixes* of elements of $\bar{U}$ as nodes, the children of a node $z$ are nodes of the form $az$, where $a \in \Sigma$, and the label of a node is its $k$-prefix. We use a DFS over the complete subtree of $\bar{\mathcal{T}}_{\bar{U}}$ rooted at $u$ and adjust the processing of a node according to the conditions listed above (Algorithm 8).

The space and time bounds are the same as for the set $U$, and the set $\bar{U}'$ of the windows with the suffix $u'$ is processed in exactly the same way as $\bar{U}$.

Adding $O(\sigma^k)$ time for finding the candidate $k$-mers $u$ and $u'$ (Algorithm 6, line 2), we get the required complexity bounds. □

**Algorithm 8** : function sf_cost for recursive count of status-changing $k$-mers

*Input:* array rank, rank $i$, candidate $k$-mers $u, u'$ to swap, current node with $k$-prefix $z$ and depth $d$, flag $f$ (set if $u'$ is already seen); global $\sigma, k, w$

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

### A.5 Proof of Theorem 6

Similar to the proof of Theorem 5, we define bad and good windows. For $v \in \Sigma^{w+k}$ and $a \in \Sigma$, the window $va$ is bad (good) for $\rho_1$ if it is charged (not charged) by $\rho_1$ while $v$ is not charged (charged) by $\rho$. By $B_1$ and $G_1$ we denote the sets of bad and good windows for $\rho_1$, respectively. Similarly, we define bad and good windows $av$ for $\rho_2$ and denote their sets by $B_2$ and $G_2$, respectively. We prove the analog of the claim from Theorem 5.

*Claim.* $|B_1| \leq |G_2|$ 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 $\rho_1$-rank (here $a, b \in \Sigma$). By the definition of $\rho_1$, $u$ has the minimum $\rho$-rank in $v$. Since $v$ is free for $\rho$, $u$ 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 $\rho$. 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 $\rho_2$-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 $\rho_2$. 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.

Let $c, c^{\leftarrow}$ be the numbers of windows charged by $\rho$ and $\rho^{\leftarrow}$, respectively. Then, $\rho_1$ charges $\sigma c + |B_1| - |G_1|$ windows, and $\rho_2$ 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.

### A.6 Running DeepMinimizer and polar sets

We failed to run DeepMinimizer [6] and polar sets [21] 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.
  ```

### A.7 Runtime and memory usage of GreedyMini+ and GreedyMiniParticular+

We ran GreedyMini+ and GreedyMiniParticular+ 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 runtime (Figure 4A) and maximum memory usage (Figure 4B) of our implementation of GreedyMini+ over various combinations of $w$ and $k$ without extending $k$ (Theorem 6).

The measured runtimes fit the bound of GreedyMini given by Theorem 3 ($O(\sigma^{2k} + w\sigma^{k+w})$). 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 GreedyMini and Swap by Theorems 3 and 4 ($O(\sigma^k + w)$). Note that due to the utilization of various libraries and other computational factors, our implementation of GreedyMini+ 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 runtime (Figure 5A) and maximum memory usage (Figure 5B) of our implementation of GreedyMiniParticular+. Over a sequence length of 1 million nucleotides, the 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 $k$.

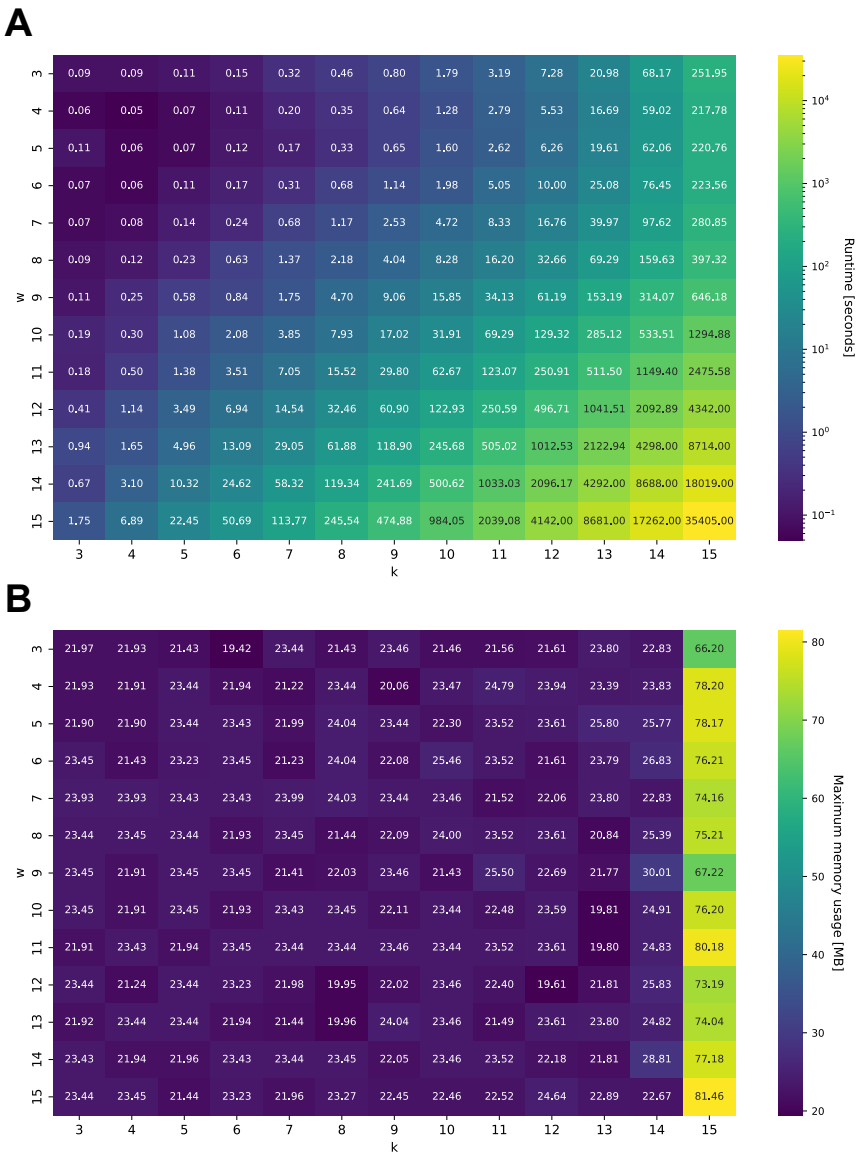


Fig. 4: Runtime (A) and maximum memory usage (B) of GreedyMini+ on various $w$ and $k$ values.

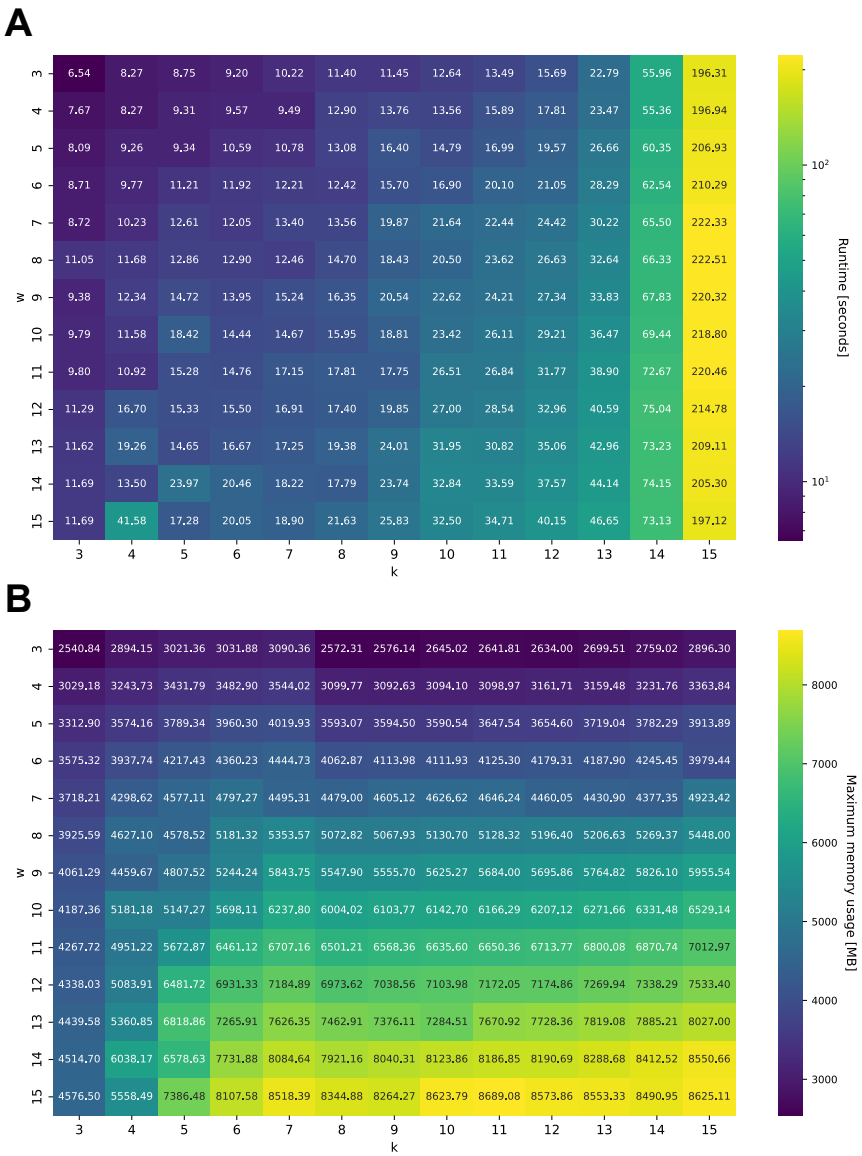


Fig. 5: Runtime (A) and maximum memory usage (B) of GreedyMiniParticular+ on various $w$ and $k$ values over 1 million nucleotides from Chromosome X.