Swiftly Identifying Strongly Unique k-Mers

Jens Zentgraf
Algorithmic Bioinformatics, Department of Computer Science,
Saarland University, Saarbrücken, Germany
Center for Bioinformatics Saar, Saarland Informatics Campus, Saarbrücken, Germany
Graduate School of Computer Science, Saarland Informatics Campus, Saarbrücken, Germany

Sven Rahmann
Algorithmic Bioinformatics, Department of Computer Science,
Saarland University, Saarbrücken, Germany
Center for Bioinformatics Saar, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

Motivation. Short DNA sequences of length k that appear in a single location (e.g., at a single genomic position, in a single species from a larger set of species, etc.) are called *unique k-mers*. They are useful for placing sequenced DNA fragments at the correct location without computing alignments and without ambiguity. However, they are not necessarily robust: A single basepair change may turn a unique k-mer into a different one that may in fact be present at one or more different locations, which may give confusing or contradictory information when attempting to place a read by its k-mer content. A more robust concept are *strongly unique k-mers*, i.e., unique k-mers for which no Hamming-distance-1 neighbor with conflicting information exists in all of the considered sequences. Given a set of k-mers, it is therefore of interest to have an efficient method that can distinguish k-mers with a Hamming-distance-1 neighbor in the collection from those that do not.

Results. We present engineered algorithms to identify and mark within a set K of (canonical) k-mers all elements that have a Hamming-distance-1 neighbor in the same set. One algorithm is based on recursively running a 4-way comparison on sub-intervals of the sorted set. The other algorithm is based on bucketing and running a pairwise bit-parallel Hamming distance test on small buckets of the sorted set. Both methods consider canonical k-mers (i.e., taking reverse complements into account) and allow for efficient parallelization. The methods have been implemented and applied in practice to sets consisting of several billions of k-mers. An optimized combined approach running with 16 threads on a 16-core workstation, yields wall-clock running times below 20 seconds on the 2.5 billion distinct 31-mers of the human telomere-to-telomere reference genome.

Availability. An implementation can be found at `https://gitlab.com/rahmannlab/strong-k-mers`.

2012 ACM Subject Classification Applied computing → Molecular sequence analysis; Applied computing → Bioinformatics; Theory of computation → Parallel algorithms; Theory of computation → Sorting and searching; Information systems → Nearest-neighbor search

Keywords and phrases k-mer, Hamming distance, strong uniqueness, parallelization, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.WABI.2024.15

Supplementary Material *Software (Source code)*: `https://gitlab.com/rahmannlab/strong-k-mers` [18], archived at `swh:1:dir:7ce51b0df8003cb2d49b99084d09f2ce6df56638`

Acknowledgements We thank Karl Bringmann for discussions about the FourWay algorithm. We also thank the four reviewers of the initial version of this manuscript for their suggestions to improve its presentation.

1 Introduction

Alignment-based sequence analysis methods are increasingly being replaced by alignment-free (or at least partially alignment-free) methods. The reason for this development is the high computational cost for sequence alignments. An early example of this development was the replacement of overlap-consensus based genome assembly by De Bruijn graph based assembly methods, subdividing the sequenced DNA fragments further into overlapping pieces of length k (so-called k-mers) [14].

Another example is transcript quantification from RNA-seq data, pioneered by kallisto [1], which assigns a transcript to each read not by computing alignments, but directly from the k-mer content of the reads. As transcripts from the same gene typically share many of the same k-mers, an elaborate procedure (based on the expectation maximization algorithm) was necessary to obtain exact read-to-transcript assignments.

More recently, there has been the example of xenograft sorting [19], where one wants to separate reads in a mixed sample from two species (typically, human tumor with surrounding mouse tissue). This had previously been done by aligning all reads to both the human genome and the mouse genome and picking the better alignment for each read to assign the species of origin. However, it is computationally much more efficient to build a k-mer index (for $k = 25$, for example) with associated species information and to classify reads according to their k-mer content. This approach can save up to 80% of the CPU work [19]. The same ideas generalize to metagenomic profiling, where one seeks to quantify the amount of different species in a metagenomic sample, again not based on alignments, but based on k-mer content [2, 7, 3].

Many more examples could be mentioned, but here we want to focus on a particular aspect of alignment-free methods: the utility of strongly unique k-mers vs. (weakly) unique k-mers vs. non-unique k-mers.

Informally, a k-mer is unique if it occurs in a single location in the sequence collection under consideration. Details depend on the definition of location, which in turn depends on the intended application. When indexing a reference genome, a k-mer is unique if it occurs at a single position on a single chromosome (considering both strands of all chromosomes). In the transcript quantification scenario, it is unique if it occurs in a single transcript (but it may occur several times in that transcript). In the xenograft or metagenomic scenario, it is unique if it occurs in a single species (possibly several times). Thus, unique k-mers are both more useful and technically easier to handle than non-unique k-mers: They give unambiguous information about the origin of the sequence that does *not* need to be further decoded or disambiguated together with information from other k-mers. They also allow us to keep the key-value store implementation simple because only a single value (location) is associated with them. In the above respective scenarios, the value would be the unique chromosome-position pair (in a reference genome), the unique transcript ID (for transcript quantification), the unique species ID (xenograft or metagenomics applications), respectively. It is also possible to consider only the k-mers without associated value, which is the setting we consider here, and where uniqueness simply means a single occurrence in the sequence collection.

For non-unique k-mers, a variable-length list of such values would need to be properly handled, which typically involves another level of indirection with more memory lookups and more cache misses. Instead, when focusing on unique k-mers, one simply stores a special value (call it ∞) for non-unique k-mers, indicating that they occur in more than one location.

Of course, the main question is whether sufficiently many unique k-mers exist in the indexed sequence collection for the intended application. For $k \geq 23$, most of the canonical k-mers in the human genome are unique [17, 11]. For transcript quantification with many similar transcripts (splice variants) from the same gene, it can be difficult to cover each transcript with sufficiently many unique k-mers (even for large values of k), but for $k \geq 25$, distinguishing between human and mouse k-mers is relatively easy, because only few k-mers occur in both species [19].

Given the advantages of unique k-mers, they should be used wherever their abundance permits, as they greatly simplify sequence data analysis. However, and this is our main point here, uniqueness is not a robust property: A single nucleotide change, such as a sequencing error or an individual single-nucleotide variant, may turn a unique k-mer x (with some associated value v) into a different k-mer x' that may either not exist at all in the indexed collection, exist as a unique k-mer with the same value v or a different value v', or as a non-unique k-mer. The critical case is that $x' \neq x$ exists as a unique k-mer with a different value $v' \neq v$, because then we receive confusing or conflicting information about the origin of the examined sequence.

Therefore, a stronger concept than just uniqueness is helpful. Consider a k-mer x and its Hamming-distance-1 neighborhood $N(x)$ containing exactly the $3k$ k-mers that differ at exactly one position from x. If none of the $x' \in N(x)$ exists in the indexed collection, then we can be certain that no single substitution can turn x into a k-mer with a different associated value. The same is true if all existing $x' \in N(x)$ have the same associated value v as x. (This idea also applies to plain sequences or multisets of k-mers without associated values.) We call such k-mers *strongly unique*. Strong uniqueness is a useful concept for alignment-free sequence analysis for the reasons stated above: If a strongly unique k-mer is seen in a sequenced DNA fragment, that fragment can be unambiguously and robustly located; a single substitution cannot give wrong information.

A technical complication arises from the double-strandedness of DNA and from the equivalence of a sequence s to its reverse complement. Formal details and definitions are therefore presented in Section 2.

The above explanation should give enough motivation to consider the *weak k-mer identification problem* (formally stated as Problem 5 in Section 2): Given a k-mer set K, identify those elements of K (called the weak k-mers) that have a Hamming-distance-1 neighbor in K. In practical settings, one combines a solution for this problem with k-mer counting to classify unique vs. non-unique k-mers and can also add comparisons of k-mer associated values, as described above.

In this work, we focus on the basic problem of weak k-mer identification, for which we present two different efficient approaches (Section 3) after giving formal definitions (Section 2). An comparative evaluation of both methods follows (Section 4), and a discussion concludes.

2 Preliminaries

We introduce several basic definitions: k-mers, their (canonical) integer encoding, bit-parallel computation of the Hamming distance between two k-mers, canonical Hamming distance, strong and weak k-mers in a set.

We only consider DNA sequences over the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ here (and in our current implementation), but the ideas generalize to other alphabets. However, we take the double-stranded nature of DNA molecules into account when considering Hamming distance between DNA sequences; this would and should not be done with different alphabets.

▶ **Definition 1** (k-mer). *Given an alphabet Σ, a* k-mer *is a sequence of length k over Σ. Given a (long) string s over Σ, a* k-mer of s *is any substring of length k of s.*

▶ **Definition 2** (reverse complement). *The* reverse complement $rc(s)$ *of a DNA sequence s is obtained by reversing the sequence and substituting `A` $\leftrightarrow$ `T` and `C` $\leftrightarrow$ `G`.*

Canonical integer encoding of k-mers. To represent and store a DNA k-mer x efficiently, it can be bijectively encoded as an integer $0 \leq enc(x) < 4^k$ for fixed k: Each base is encoded as a number in $\{0, 1, 2, 3\}$ (e.g. lexicographically), and the resulting sequence of k numbers is interpreted as a base-4 integer. Equivalently, and relevant for the bit-parallel Hamming distance test described below, we may write the same integer in its $2k$-bit representation. For example, $enc(\texttt{TACG}) = (3012)_4 = (11|00|01|10)_2 = 198$.

To ensure that a k-mer x and its reverse complement $rc(x)$, which both represent the same DNA molecule, are encoded by the same integer value, we define the *canonical integer encoding* or *canonical code* of x as $cc(x) := \max\{enc(x), enc(rc(x))\}$. (In most of the literature, the definition is given with the minimum instead of the maximum, but it does not matter in the context here. The maximum-based definition offers advantages when also using minimizers to avoid taking a minimum twice.) For example, $cc(\texttt{TACG}) = cc(\texttt{CGTA}) = \max\{enc(\texttt{TACG}), enc(\texttt{CGTA})\} = \max\{198, 108\} = 198$.

Hamming distance between two k-mers. The Hamming distance $d(x, y)$ between two k-mers x, y is the number of positions in which x and y differ. For 2-bit encoded DNA sequences, there is a fast bit-parallel way to compute the Hamming distance and to test whether $d(x, y) \leq 1$.

From the $2k$-bit patterns of x and y – let them be $p = (p_{2k-1}, \ldots, p_0)$ and $q = q_{2k-1}, \ldots, q_0$, respectively – we compute a bit pattern $h = (h_{2k-1}, \ldots, h_0)$ that has $h_i = 0$ for all odd i, and $h_i = 1$ for even i if and only if the nucleotides encoded by (p_{i+1}, p_i) and (q_{i+1}, q_i) differ.

To achieve this, we first compute $u := p \oplus q$, where $\oplus$ is the bitwise exclusive-or (XOR) operation, setting those bits $u_i = 1$ where $p_i \neq q_i$. We then combine the two bits of each nucleotide into the even bits of u and clear the odd bits to indicate which nucleotides differ between x and y by setting $h := (u \,|\, (u \gg 1)) \,\&\, (0101\ldots01)_2$. Here, the operators $|$, $\&$ and $\gg$ represent bitwise or, bitwise and, and bit shift right, respectively. The obtained h has the desired properties. The population count (number of 1-bits) of h then equals the Hamming distance between the k-mers x and y.

Testing whether the Hamming distance is at most 1 (i.e., whether h is zero or a power of 2) can be further simplified by computing $w := h \,\&\, (h - 1)$. We have $w = 0$ if and only if h has at most a single 1-bit, and $w \neq 0$ if and only if the Hamming distance between the k-mers represented by x and y is at least 2.

Canonical Hamming distance. In the following, we interpret DNA k-mers as double stranded molecules, i.e., both x and $rc(x)$ are represented by either of them (in practice, their canonical integer encoding $cc(x)$). This has to be considered for Hamming distance computations.

For example, take x = `AAAA` and y = `ATTT`. Seen as single-stranded k-mers, clearly $d(x, y) = 3$. However, seen as double-stranded DNA molecules, `ATTT` is equivalent to its reverse complement `AAAT`, and $d(x, rc(y)) = 1$. Therefore, we make the following definition.

▶ **Definition 3** (Canonical Hamming distance). *Given DNA k-mers x, y, their* canonical Hamming distance *is*

$$H(x,y) := \min\{d(x,y), d(x, rc(y))\}\,. \tag{1}$$

Problem Statement. As stated in the introduction, our goal is to identify the strongly unique canonical k-mers in a collection of sequences. The uniqueness property is identified by k-mer counting, for which there exist many efficient methods and tools with different strength and weaknesses, such as KMC2 [4], KMC3 [8], Gerbil [6], Jellyfish [10], hackgap [20], Kaarme[5], among others. In fact, being able to count up to 2 is sufficient for the present purpose. The output of a k-mer counter is a set K of distinct canonical k-mers, and a count value for each $x \in K$. The missing piece is to identify the weak canonical k-mers in K in the following sense.

▶ **Definition 4** (Weak k-mer, strong k-mer). *Given a set K of distinct canonical DNA k-mers, a canonical k-mer $x \in K$ is called* weak *(within K) if there exists another $y \in K$ with $H(x,y) = 1$. The other canonical k-mers are called* strong *(within K).*

When we mark weak canonical k-mers in K and additionally have their counts, we have identified the strongly unique canonical k-mers as those with a count of 1 that are not weak in K. Therefore, we here focus on the following problem.

▶ **Problem 5** (Weak canonical k-mer identification). *Given a set K of canonical DNA k-mers, identify the subset $W \subseteq K$ of the weak canonical k-mers.*

How a solution to this problem is implemented in a concrete setting may vary with the representation of K. In practice, the elements of K will typically be canonical integer codes of k-mers, and we may use one more bit to indicate the weak ones.

We point out two obvious algorithms (that are too inefficient in practice) to solve the problem. Let K contain n distinct k-mers.

1. Full pairwise comparison: Use the bit-parallel method from above to test all $\binom{n}{2} = n(n-1)/2$ pairs $x \neq y$ whether they satisfy $H(x,y) \leq 1$, and if yes, mark both x and y as weak.
2. Neighborhood generation: If K is given as a hash table allowing efficient look-up, for each $x \in K$, generate the canonical integer codes of the $3k$ Hamming-distance-1 neighbors, and look them up in K. If any neighbor is found, mark both x and the discovered neighbor as weak.

The first method is impractical because of its quadratic running time. The second method has a running time linear in n, but with a factor of $3k$, and every memory lookup is likely a cache miss. It is thus slow in practice, but can still be useful as a baseline for comparison. In the following section, we present more efficient practical methods for identifying weak canonical k-mers. An evaluation follows in Section 4.

3 Methods

We present two algorithms to identify weak k-mers. Independently of the given form of input (hash table based key-value store, or sorted or unsorted list of distinct k-mers) and independently of the used algorithm, we first create a *lexicographically sorted* list of the k-mers *and* their reverse complements. If the original input consists of canonical k-mers, or the reverse complements are not included for another reason, this step may double the input size when we add the reverse complements explicitly. This step is necessary for both algorithms to ensure that we consider the canonical Hamming distance instead of the (standard) Hamming distance for the original input.

The first algorithm is based on 4-way comparisons, similar to a multi-way mergesort. In the first round, the sorted k-mer list is divided into 4 buckets based on the k-mers' first nucleotide (A, C, G, T). For each nucleotide c, a pointer p_c initially points at the start of the c-bucket. In each step, we compare the four (or possibly less, once entire buckets have been processed) elements pointed to by the p_c, $c \in \Sigma$, ignoring the bucket prefixes. (In the first round, the bucket prefixes are single nucleotides, so we compare only the remaining $(k-1)$-mers.) We identify the set $C^* \subseteq \Sigma$ such that the p_c, $c \in C^*$, point to the minimal element(s) of the examined $(k-1)$-mers. If $|C^*| = 1$, there is a unique minimal $(k-1)$-mer, but if $|C^*| \geq 2$, we have found equal minimal $(k-1)$-mers, i.e. k-mers that differ only in their first nucleotide, and therefore a group of weak k-mers. The p_c, $c \in C^*$, are incremented, and the process is repeated until all buckets have been fully processed. In the next round, each of the buckets is subdivided into 4 sub-buckets, and the 4-way comparison is repeated recursively for each sub-bucket (with k decreased by 1). Details are given in Section 3.1.

The second algorithm does pairwise Hamming distance tests, but on small buckets. If $H(x, y) = 1$, then the one difference must be either in the first, second, third, or fourth quarter of the sequence (rounding arbitrarily for now). Because the input contains all reverse complements, it is sufficient to consider the third and the fourth quarter: If x, y differ in the first (second) quarter, then $rc(x), rc(y)$ differ in the fourth (third) quarter, respectively. If the difference is in the fourth quarter, the first $3k/4$ nucleotides are equal, and we can linearly scan the sorted k-mer list, divide it into corresponding buckets of equal $3k/4$ first nucleotides and run a bit-parallel pairwise Hamming test within each bucket. The overall efficiency of this approach depends on the relation between n and $4^{3k/4}$ and not having large buckets. If k is sufficiently large, many of the buckets will contain only a single or two elements, and the process is indeed very fast. It remains to deal with pairs x, y that differ in one nucleotide in their third quarter. Details are given in Section 3.2.

3.1 Recursive 4-way comparisons (FourWay)

Basic algorithm. The algorithm FourWay is based on 4-way merges. The input is a set K of canonical k-mers, which are "unpacked" into twice as many forward and reverse complement k-mers, which are then sorted lexicogaphically. This yields a sorted k-mer array A of length n.

The algorithm FourWay is recursive. It is called with a depth parameter $d \in \{1, \dots, k-1\}$ and a list of start pointers $q = (q_c)$ with $c \in \Sigma \cup \{\$\}$. Each invocation FourWay(A, d, q) identifies weak k-mers in an interval $I = [q_{\mathtt{A}}, q_{\$}[$ of A, where the first $(d-1)$ nucleotides of the contained k-mers are equal, and q_c (for $c \in \Sigma$) points to the start of the sub-interval of I where the d-th nucleotide is c (Figure 1). The sentinel pointer $q_{\$}$ points beyond the end of the interval I. In the initial call, $d = 1$ (no common prefix, the first nucleotide is compared), $q_{\mathtt{A}} = 0$ points to the first (smallest) element in A (with indexing starting at zero) and $q_{\$} = n$ points past the end of A. The initial values of the other pointers q_c, $c \in \{\mathtt{C}, \mathtt{G}, \mathtt{T}\}$ are determined by linearly scanning the sorted array once.

First, we create working copies $p = (p_c)$ of $q = (q_c)$ for $c \in \Sigma$. While q will stay unchanged, the p_c increase towards larger elements as the algorithm proceeds. Initially, all the p_c pointers are *active*. When p_c moves beyond the end of its sub-interval (i.e., it reaches a $q_{c'}$ for a character $c' > c$, where the sentinel is larger than any character in Σ), the pointer becomes *inactive*. If only a single pointer or no pointer is active any more, we are done and proceed to the recursive calls; see below.

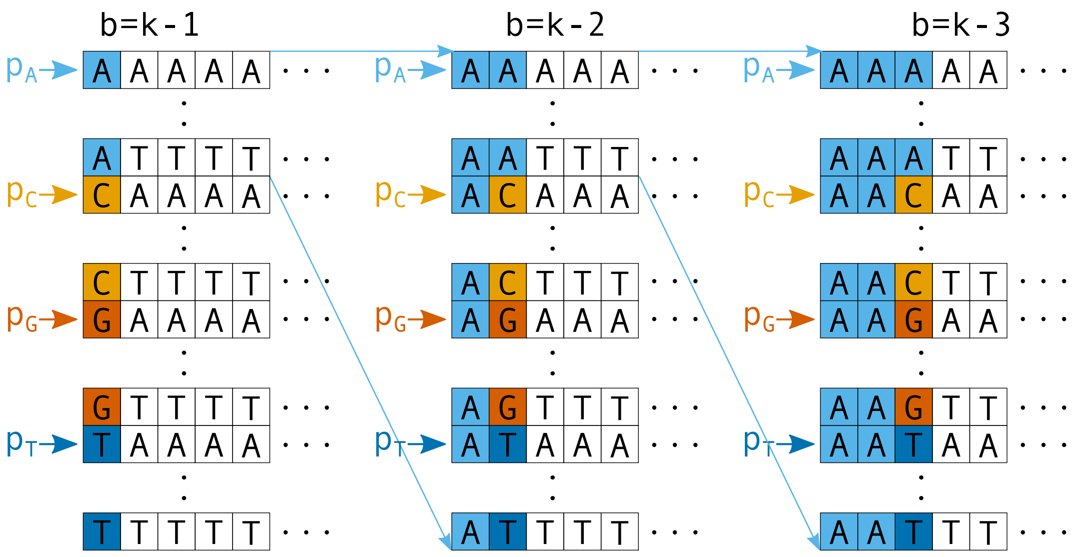

Figure 1 Recursive 4-way comparison at different depths $d = 1, 2, 3$, from left to right. At depth d, the first $d - 1$ characters of all k-mers within the considered interval are equal, and the d-th character is compared. Let $b := k - d$. The (at most) four b-mer suffixes of the k-mers pointed to by pointers p_c, $c \in \Sigma$ are compared, and the minimal b-mer(s) are identified among the four elements. The k-mers with equal minimal b-mers are marked as weak if there are at least two equal minimal b-mers. The pointers of minimal b-mers are incremented (or inactivated, if their bucket has been completely scanned). After the 4-way comparison of an interval is completed, it is repeated recursively with increased depth $d + 1$ on its 4 sub-intervals (only the first such recursive call for the `A`-subinterval is shown).

In each step, the algorithm examines the k-mers at the locations pointed to by the active pointers p_c. We call these the *active k-mers*. They jointly have the following properties:

1. Their first $d - 1$ characters are equal (true for all k-mers in interval I),
2. their d-th characters are distinct,
3. their suffixes of length $b = k - d$ are arbitrary, but examined in increasing order;
4. they are the smallest k-mers in I that have not yet been but still may be identified as weak based on a single difference at their d-th character.

The comparison is visualized in Figure 2. We look at the b-suffixes of the active k-mers and find the smallest one(s). Let C^* be the character set such that exactly the k-mers at p_c, $c \in C^*$, have the minimal b-suffixes among the active k-mers. If $|C^*| \geq 2$, we have identified a group of k-mers of size $|C^*|$ that differ only at their d-th position; hence all of them are marked as weak. (If $|C^*| = 1$, nothing happens.) Then, all p_c for $c \in C^*$ are incremented. These steps are repeated until a single (or no) active p_c remains.

After processing interval I, if $d < k$, the sub-intervals are processed recursively, so there are $|\Sigma| = 4$ recursive calls, each with increased $d \leftarrow d + 1$ (and reduced $b = k - d$). The initial pointers q for each subinterval are obtained by a linear scan through the sub-interval. The recursive call is not performed if the length of the subinterval is at most 1, as there is nothing to compare then.

Implementation on $2k$-bit encoded integers. The current implementation uses a $2k$-bit encoded representation of k-mers and is restricted to $k \leq 31$, leaving at least one bit for marking weak k-mers within the 64-bit integer encoding.

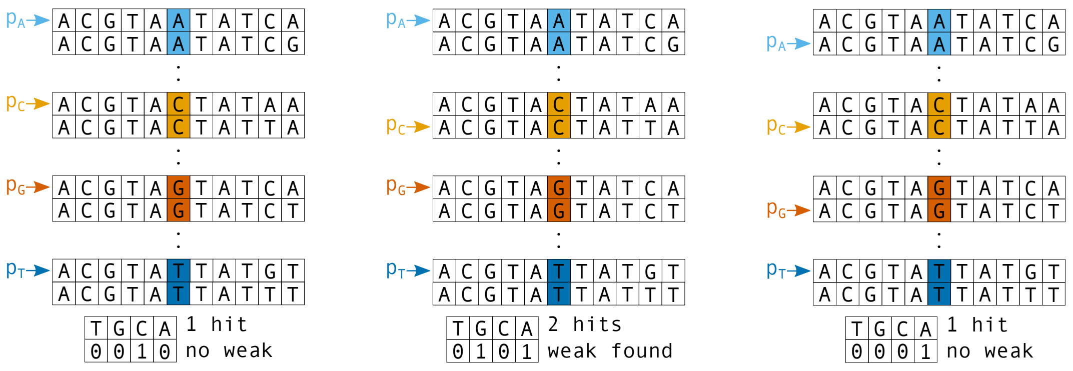

Figure 2 Weak k-mer identification by 4-way comparison at depth $d = 6$. The first $d - 1 = 5$ characters are identical. Pointers $p_c, c \in \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$, point at k-mers whose d-th character is c. Considering the $b := (k - d)$-suffixes, we identify the smallest suffix among the pointed-to k-mers. **Left panel:** The smallest suffix is `TATAA` at $p_\mathtt{C}$, as indicated by the bit vector (0010). As a single b-suffix is minimal, no weak k-mers are identified, and $p_\mathtt{C}$ is increased. **Middle panel:** The smallest b-suffix is `TATCA` at $p_\mathtt{A}$ and $p_\mathtt{G}$. Therefore, the two k-mers pointed to by $p_\mathtt{A}$ and $p_\mathtt{G}$ are identical except for their d-th characters, and we have found a weak pair. Both $p_\mathtt{A}$ and $p_\mathtt{G}$ are incremented. **Right panel:** The smallest b-suffix is `TATCG` at $p_\mathtt{A}$ only. No weak k-mers are identified, and $p_\mathtt{A}$ is incremented.

In order to compute C^*, and to keep track of the smallest k-mers, a 4-bit vector v is used. Iterating over the active k-mers only, if the b-suffix is a new minimum (and also initially), a single bit is set in v, corresponding to the nucleotide at position d (using $\mathtt{A} = (0001)_2 = 1$, $\mathtt{C} = (0010)_2 = 2$, $\mathtt{G} = (0100)_2 = 4$, $\mathtt{T} = (1000)_2 = 8$). If another b-suffix is equal to the current minimum, the bit corresponding to the character at position d is additionally set in v (using bitwise or).

We test whether $|C^*| \geq 2$ by checking if the population count of v is at least 2, which is done by testing if $v \,\&\, (v - 1) \neq 0$.

Optimizations and parallelization. In principle, the depth d is recursively increased up to k. However, since the array contains both forward and reverse-complement k-mers, we identify each pair with a Hamming distance of 1 twice, both as x, y and as $rc(x), rc(y)$. In one case, the difference is in the first half; in the other case, the difference is in the second half of the k-mers. Therefore, we could stop the recursion at depth $\lceil k/2 \rceil$.

However, it is even better to process only the second half of the k-mers, i.e. start at depth $d = \lfloor k/2 \rfloor$. By an initial linear scan, we identify intervals whose k-mers share the first $(d - 1)$ nucleotides and divide them among parallel threads, as each such interval can be processed independently. This allows for almost trivial parallelization among many threads. In addition, the processed intervals at $d \geq k/2$ tend to be short already and can be handled with good cache locality.

As the depth d approaches k, the intervals become shorter and shorter, even singletons or empty. Even before that point, the book-keeping required by the recursion creates more work than the actual comparison of elements. Therefore, if the interval length drops below a certain value (we determined 24 as a reasonable threshold), we switch to the direct bit-parallel pairwise comparison, which has excellent cache locality on short intervals. We call this optimization the FourWay+Pairwise method.

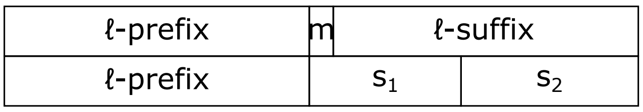

Figure 3 All k-mers are split into blocks based on the $\ell := \lfloor k/2 \rfloor$-prefix and further separated into sections based on third quarter s_1 and last quarter s_2.

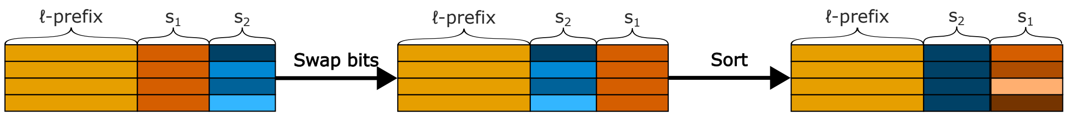

Figure 4 After checking if the k-mers differ in a single position in the last quarter s_2, existence of a single difference in the third quarter s_1 has to be checked. For this, the nucleotides of s_1 and s_2 are swapped. Then, k-mers are locally re-sorted (within buckets of common $\ell = \lfloor k/2 \rfloor$-prefixes), and the algorithm for the last quarter is applied again.

Overall, the FourWay and FourWay+Pairwise algorithms mainly perform (4-way) linear scans of intervals of the array, and therefore have excellent cache locality. Hardware prefetching takes care of moving required k-mers into the CPU caches before they are needed for comparison, so there is very little memory latency and high memory throughput for these algorithm by design.

3.2 Pairwise comparisons in small buckets (Quarter)

With the Quarter algorithm, we reduce the number of pairs for which we calculate the Hamming distance. If two canonical k-mers $x \neq y$ have a canonical Hamming distance of $H(x, y) = 1$, then at least one of the pairs from $\{x, rc(x)\} \times \{y, rc(y)\}$ have their single difference in the third quarter s_1 or last quarter s_2 of their sequences; in the case of odd k, the middle position m must be included in the third quarter s_1 (see Figure 3).

If the difference is in the last quarter s_2, then pairwise comparisons can be restricted to within buckets that share a common $\ell := \lfloor k/2 \rfloor$-prefix and a common s_1 section (for an overall shared $3k/4$-prefix of the k-mer). For example, for $k = 25 = (6 + 6 + 7 + 6)$, we partition the sorted k-mer array into intervals that share the same $(6 + 6 + 7) = 19$-mers. As already many 19-mers are unique in a mammalian genome, the 25-mer intervals are often small or even contain just a single 25-mer, requiring no further comparisons at all.

If the difference is in the third quarter s_1, then pairwise comparisons can be restricted to sets of k-mers that share both their ℓ-prefix and the sequence in the last quarter s_2. These sets may be conveniently constructed by local re-sorting within blocks that share a common ℓ-prefix: Swap the nucleotides belonging to sections s_1 with those in s_2 and re-sort locally within the ℓ-block (Figure 4); then apply the same interval partitioning as above (in the example using common $(6 + 6 + 6) = 18$-mers.

Implementation and parallelization. To keep track of which k-mers are marked as weak, we use one of the 64 bits of the k-mers. This limits the implementation to $k \leq 31$. For the Quarter algorithm, we use the least significant bit for marking weak k-mers. In the sorting step after swapping s_1 and s_2 it is not necessary to special case the bit. Since it is the least significant bit, the k-mers are sorted correctly.

The algorithm can be easily parallelized over the initial $\ell = \lfloor k/2 \rfloor$-mers, as these subintervals may be processed independently; this is similar to the FourWay algorithm.

3.3 Memory usage and chunking

As presented here, the algorithms start with an already sorted k-mer set K that contains both forward and reverse-complemented k-mers. In practice, one more often has an unsorted collection of canonical k-mers available, e.g., from a key-value store such as a hash table or file on disk.

However, it may take considerable time and additional memory to convert the available representation into the required input of the algorithms presented here. For example, the k-mer counter hackgap can represent the roughly 2.39 billion distinct human canonical 25-mers and counts up to 255 in less than 12 GB of memory, using bit packing and quotienting $\times$[20]. However, expanding these into twice as many (4.78 billion) 64-bit integers takes an additional 38 GB of memory for identifying the weak 25-mers.

On smaller-memory systems, the input data can be created and processed in smaller chunks (at the expense of speed): Select a small number $s \leq \lfloor k/2 \rfloor$ of initial nucleotides; often $s = 1$ or $s = 2$ is sufficient in practice. Split up the input set K into 4^s chunks, K_j, $j = 0, \ldots, 4^s - 1$, where chunk K_j is defined as the sorted subset of K that has an integer-encoded length-s prefix with value j (i.e., $enc(x[: s]) = j$). Each chunk is generated by linearly scanning over the existing representation and extracting only k-mers and reverse complements that start with the correct prefix; this is done 4^s times. The default case (everything is one big chunk) corresponds to $s = 0$. All presented algorithms can be run on each chunk sequentially (and then parallelized based on sub-chunks based on more initial characters).

3.4 Result interpretation

The presented algorithms only identify one pair with canonical Hamming distance 1, say $x \neq y$ with their difference in the second half of the sequences, but not the other pair, $rc(x) \neq rc(y)$, with their difference in the first half of the sequences. It is this property that gives us chunks and parallelization essentially for free.

However, this means that post-processing is required. This is typically required anyway, as the weak k-mers must be annotated in the original (canonical) representation (say, the hash table). This involves a linear scan of the annotated sorted set K and a look-up of the canonical form in the original representation.

In the following evaluation, (times for) pre- and post-processing are not considered. However, a practical use-case for xenograft sorting is presented, including pre- and post-processing times (Section 4.3).

4 Computational Experiments

We evaluate FourWay, FourWay+Pairwise and Quarter on the human telomere-to-telomere (t2t) reference genome [13] with roughly 3.1 billion basepairs and 2.5 billion distinct 31-mers. After presenting results on the exact number of weak and strong k-mers for different values of k, we compare the algorithms' running times and their speedup factors when increasing the number of threads.

Our implementations are written in `numba`-compiled Python 3.11 [9]. We read a pre-computed sorted `numpy` array [16] from disk. The benchmarking equipment consists of a workstation with AMD Ryzen 9 5950X 16-core processor with 128 GB of main memory. Code is available at `https://gitlab.com/rahmannlab/strong-k-mers`.

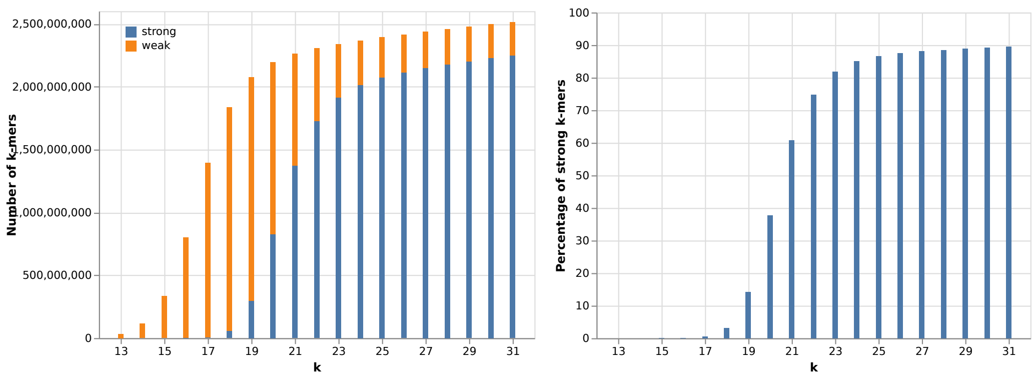

Figure 5 Left: Number of strong and weak k-mers in the t2t reference genome for different values of k (stacked bar chart). The total number of k-mer rapidly increases for k between 13 and 19. For $k \geq 21$, there are only a few new k-mers with increasing k. **Right:** The percentage of strong k-mers among all distinct k-mers. For $k \geq 25$, almost 90% of the distinct k-mers are strong.

The array mentioned above is pre-computed as follows: We compute the set of all canonical k-mers by executing the k-mer counter hackgap [20]. This results in an in-memory hash table with all canonical k-mers (encoded as $2k$-bit integers) and their counts. The canonical k-mers are expanded to 64-bit integer encodings of the k-mers and their reverse complements, and stored in a large `uint64` numpy array, which is then sorted and written to disk. The time for this preprocessing step is not included in the measurements reported here.

4.1 Number of strong and weak k-mers

As shown in Figure 5, with increasing k, the total number of distinct k-mers in the t2t reference genome increases. Since the t2t reference genome contains ≈ 3.1 billion nucleotides, for $k \leq 16$, most of the approximately $4^k/2$ canonical k-mers are present in the sequence. Consequently, almost all of these k-mers are weak. For $k \geq 18$, the k-mers get more unique and specific, and the increase of distinct k-mers in the sequence flattens out for $k \geq 21$. At the same time, the number of strong k-mers increases. For $k = 18$, only approximately 4% of the k-mers are strong. For $k = 23$, already more than 80% of the k-mers are strong. For $k \geq 24$, the number of strong k-mers gradually approaches 90% of the distinct k-mers.

4.2 Running Times

We compare the running times of our implementations of algorithms FourWay, FourWay+Pairwise and Quarter (Figure 6 left). As a baseline, the neighborhood generation method (looking up all $3k$ canonical neighbor k-mers, or less until the first neighbor is found, for each canonical k-mer in the input set; skipping k-mers already marked as weak) for $k = 25$ using 8 threads needs 3937 seconds (approximately 65 minutes), using the multi-way Cuckoo hash tables of hackgap [20].

For $k \in \{13, \ldots, 31\}$, each algorithm was executed three times; individual running times and their mean are shown in Figure 6 (left).

The running time of FourWay correlates with the total number of distinct k-mers. For $k \leq 22$, the running time increases up to approximately 140 seconds. The running time for $k \geq 22$ remains nearly constant at 140 seconds.

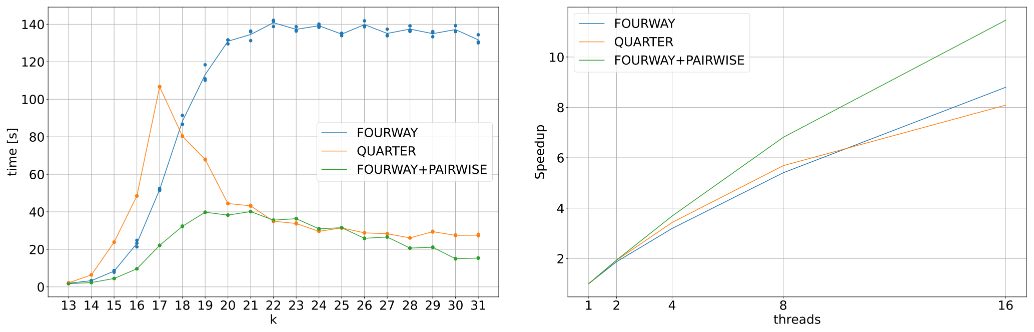

Figure 6 Comparison of the running time of FourWay, FourWay+Pairwise and Quarter. **Left:** running time for different values of k using 8 threads. The naive neighborhood generation approach needs 3937 seconds ($\approx$ 65 min) for $k = 25$ using 8 threads, while all presented algorithms need less than 140 seconds, the best ones around 30 seconds in this case. (For a fair comparison to neighborhood generation, the pre- and post-processing time of about 300 seconds should be added.) **Right:** Speedup as a function of the number of threads used for parallelization, measured on a AMD Ryzen 9 5950X 16-core processor, for a fixed value of $k = 25$.

The optimization of FourWay+Pairwise, stopping the recursion early if the interval to be examined has at most 24 elements (276 cache-friendly pairwise comparisons), yields a significant reduction of running time, especially for large k. Using 8 threads, the longest running time is 40 seconds for $19 \leq k \leq 21$. For longer k-mers, the running time decreases down to below 20 seconds for $k = 31$.

The running time of Quarter increases with the number of distinct k-mers for $k \leq 17$. Interestingly, the algorithm has its highest running time for $k = 17$ with approximately 110 seconds, even though the number of distinct k-mers increases further for larger k, while the running time decreases. We here start to see the benefit of longer and more specific $3k/4$ ($\geq$ 13-mer) prefixes that for increasing k yield smaller and smaller buckets for the quadratic-time comparison. If k is increased further, this effect is even more noticeable, as the buckets become smaller, while the total number of k-mers stays approximately constant. For $k \geq 24$, the running time is nearly constant at approximately 30 seconds.

Thus, for $k \geq 24$, Quarter is approximately 4.5 times faster than the purely recursive FourWay approach. However, the optimized FourWay+Pairwise approach, where the recursion stops early and intervals of length at most 24 are examined with the bit-parallel pairwise Hamming distance computation, is even faster than Quarter for $k \geq 28$ and for $k \leq 19$ and comparable to Quarter in-between ($20 \leq k \leq 27$).

To compare these times fairly to the baseline neighborhood generation method, one should add 300 seconds of pre- and post-processing time to the times shown in Figure 6. Nevertheless, this yields 330 seconds for Quarter and FourWay+Pairwise against almost 3400 seconds for the neighborhood generation approach.

Parallelization. We examine the effect of parallelization for an increasing number of threads (Figure 6, right), using a constant value of $k = 25$ and 1, 2, 4, 8 and 16 threads. We compute the speedup for T threads as usual, dividing the time used by a single thread by the time used by T threads, which ideally would give a ratio of T. We see that for two threads, the speedup is nearly the desired factor of 2 for all algorithms. For $T = 4$ threads, the speedup for FourWay and Quarter is closer to 3 than to 4, but for FourWay+Pairwise, it reaches almost 4. For 8 threads, the result is similar: a speedup of roughly 6 emerges for FourWay and QUARTER, but almost 7 for FOURWAY+PAIRWISE. The effect is even more pronounced for 16 threads, where QUARTER achieves a speedup of 8, FOURWAY a speedup of nearly 9, but FOURWAY+PAIRWISE yields a speedup of almost 12. Overall, the parallelization scales quite well for all algorithms, with a distinct advantage for FOURWAY+PAIRWISE.

4.3 Use case: xenograft sorting

The alignment-free xenograft sorting tool `xengsort` [19] initially builds an index of all canonical 25-mers in the human and mouse reference genome and associates species information (only human, only mouse, both) with each canonical 25-mer. It then runs a method to identify the weak canonical k-mers (k-mers that occur in human or both and have a neighbor in mouse or both and vice versa) on the $4\,496\,607\,845 \approx 4.5$ billion 25-mers in the union of the genomes. This takes 158 wall-time *minutes* overall, even though it is (at least partially) parallelized using 8 threads, as reported in the original publication [19]. The used method is similar to QUARTER, but does quadratic-time pairwise search on buckets that share only a $\lfloor k/2 \rfloor$-prefix. One has to note that this reported time on a 64 GB machine does includes several chunk passes through the array and pre- and post-processing.

We have replaced this method in `xengsort` by QUARTER and run it under the same circumstances on the same machine, with the same number of passes, the same number of threads, and with the same pre- and postprocessing. The total time using the QUARTER algorithm is 6.1 minutes, a speedup factor of 25.9.

5 Discussion and Conclusion

In this work, we have introduced the weak canonical k-mer identification problem (as a step to identify strongly unique k-mers). This problem does not seem to have been considered at large scale elsewhere, even though the identification of strongly unique k-mers is useful in many different contexts in alignment-free sequence analysis. We have developed and evaluated three engineered algorithms to identify weak k-mers. They require a sorted array of k-mers and their reverse complements as input.

The best method FOURWAY+PAIRWISE needs at most 40 seconds to identify weak k-mers for any $k = 13, \ldots, 31$ on the human t2t reference genome, using 8 threads. All methods need less than 2.5 minutes for each of these tasks. These times are much faster than for querying all $3k$ canonical neighbors for each canonical k-mer in a fast hash table (65 minutes for 25-mers on the same dataset), even when adding the initial sorting time for the k-mer array (below 300 seconds).

The FOURWAY recursive comparison based algorithm benefits from its cache-friendly design and from hardware prefetching, but suffers from book-keeping overhead once the examined intervals become small. The improved FOURWAY+PAIRWISE method that switches to pairwise Hamming distance tests on small intervals is always faster (using an interval length threshold of 24). The QUARTER grouping algorithm shows comparably good performance to FOURWAY+PAIRWISE for $k \geq 20$, but suffers from large buckets for smaller k. Overall, FOURWAY+PAIRWISE is the method of choice with overall best performance and excellent parallelization performance.

In future work, one might attempt to further optimize the generated code for each algorithm, e.g., removing branches in the code which may lead to branch mispredictions that cost time. However, since the code generation is handled by `numba` and the LLVM compiler infrastructure [9], the relations between the written Python code and the finally generated optimized code is non-obvious.

It would be of high interest to develop methods that do not require a sorted expanded array of k-mers as input but that instead work directly on a compact hash table representation of the input set, similar to neighborhood generation, but with performance comparable to the methods presented here.

The algorithms in this work have been specially engineered for a Hamming distance of 1. It would be interesting to extend the problem for Hamming distances ≥ 2 from a practical point of view, as this would increase the tolerance against a combination of sequencing errors and SNPs in a small interval. However, one would most likely need a different approach, as a transformation to eulertigs [15] and branching on their FM index, or an approach based on search schemes [12].

References

1 Nicolas L Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic rna-seq quantification. *Nature biotechnology*, 34(5):525–527, 2016.

2 Florian P Breitwieser, Daniel N Baker, and Steven L Salzberg. KrakenUniq: confident and fast metagenomics classification using unique k-mer counts. *Genome biology*, 19(1):198, 2018.

3 C. Titus Brown and Luiz Irber. sourmash: a library for minhash sketching of DNA. *Journal of Open Source Software*, 1(5):27, 2016. doi:10.21105/joss.00027.

4 S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, May 2015.

5 Diego Díaz-Domínguez, Miika Leinonen, and Leena Salmela. Space-efficient computation of k-mer dictionaries for large values of k. *Algorithms for Molecular Biology*, 19(1):14, 2024.

6 M. Erbert, S. Rechner, and M. Müller-Hannemann. Gerbil: a fast and memory-efficient k-mer counter with GPU-support. *Algorithms Mol Biol*, 12:9, 2017.

7 Pascal Hirsch, Leidy-Alejandra G Molano, Annika Engel, Jens Zentgraf, Sven Rahmann, Matthias Hannig, Rolf Müller, Fabian Kern, Andreas Keller, and Georges P Schmartz. Mibianto: ultra-efficient online microbiome analysis through k-mer based metagenomics. *Nucleic Acids Research*, page gkae364, 2024.

8 M. Kokot, M. Dlugosz, and S. Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, September 2017.

9 Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based python JIT compiler. In Hal Finkel, editor, *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*, pages 7:1–7:6. ACM, 2015. doi:10.1145/2833157.2833162.

10 G. Marcais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, March 2011.

11 Sven Rahmann, Marcel Martin, Johannes H. Schulte, Johannes Köster, Tobias Marschall, and Alexander Schramm. Identifying transcriptional miRNA biomarkers by integrating high-throughput sequencing and real-time PCR data. *Methods*, 59(1):154–163, January 2013.

12 Luca Renders, Lore Depuydt, Sven Rahmann, and Jan Fostier. Automated design of efficient search schemes for lossless approximate pattern matching. In Jian Ma, editor, *Research in Computational Molecular Biology - 28th Annual International Conference, RECOMB 2024, Cambridge, MA, USA, April 29 - May 2, 2024, Proceedings*, volume 14758 of *Lecture Notes in Computer Science*, pages 164–184. Springer, 2024. doi:10.1007/978-1-0716-3989-4_11.

13 Arang Rhie, Sergey Nurk, Monika Cechova, Savannah J Hoyt, Dylan J Taylor, Nicolas Altemose, Paul W Hook, Sergey Koren, Mikko Rautiainen, Ivan A Alexandrov, et al. The complete sequence of a human Y chromosome. *Nature*, 621(7978):344–354, 2023.

14 Raffaella Rizzi, Stefano Beretta, Murray Patterson, Yuri Pirola, Marco Previtali, Gianluca Della Vedova, and Paola Bonizzoni. Overlap graphs and de Bruijn graphs: data structures for de novo genome assembly in the big data era. *Quant. Biol.*, 7(4):278–292, 2019. doi:10.1007/S40484-019-0181-X.

15 Sebastian S. Schmidt and Jarno N. Alanko. Eulertigs: minimum plain text representation of k-mer sets without repetitions in linear time. *Algorithms Mol. Biol.*, 18(1):5, 2023. `doi:10.1186/S13015-023-00227-1`.

16 Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy array: A structure for efficient numerical computation. *Comput. Sci. Eng.*, 13(2):22–30, 2011. `doi:10.1109/MCSE.2011.37`.

17 N. Whiteford, N. Haslam, G. Weber, A. Prügel-Bennett, J. W. Essex, P. L. Roach, M. Bradley, and C. Neylon. An analysis of the feasibility of short read sequencing. *Nucleic acids research*, 33(19):e171, 2005. `doi:10.1093/nar/gni170`.

18 Jens Zentgraf and Sven Rahmann. Identification of strongly unique k-mers. Software, version 1.0., swhId: `swh:1:dir:7ce51b0df8003cb2d49b99084d09f2ce6df56638` (visited on 2024-08-12). URL: `https://gitlab.com/rahmannlab/strong-k-mers`.

19 Jens Zentgraf and Sven Rahmann. Fast lightweight accurate xenograft sorting. *Algorithms Mol. Biol.*, 16(1):2, 2021. `doi:10.1186/S13015-021-00181-W`.

20 Jens Zentgraf and Sven Rahmann. Fast gapped k-mer counting with subdivided multi-way bucketed cuckoo hash tables. In Christina Boucher and Sven Rahmann, editors, *22nd International Workshop on Algorithms in Bioinformatics, WABI 2022, September 5-7, 2022, Potsdam, Germany*, volume 242 of *LIPIcs*, pages 12:1–12:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPICS.WABI.2022.12`.