Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

OXFORD

Subject Section

Seeding with Minimized Subsequence

Xiang Li [1], Qian Shi [1,†], Ke Chen [1,†], and Mingfu Shao [1,2,*]

[1]Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA
[2]Huck Institutes of the Life Sciences, The Pennsylvania State University, University Park, PA 16802, USA

[†]Contribute equally to this work.
[*]To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: Modern methods for computation-intensive tasks in sequence analysis (e.g., read mapping, sequence alignment, genome assembly, etc.) often first transform each sequence into a list of short, regular-length seeds so that compact data structures and efficient algorithms can be employed to handle the ever-growing large-scale data. Seeding methods using kmers (substrings of length k) have gained tremendous success in processing sequencing data with low mutation/error rates. However, they are much less effective for sequencing data with high error rates as kmers cannot tolerate errors.
Results: We propose SubseqHash, a strategy that uses subsequences, rather than substrings, as seeds. Formally, SubseqHash maps a string of length n to its smallest subsequence of length k, $k < n$, according to a given order over all length-k strings. Finding the smallest subsequence of a string by enumeration is impractical as the number of subsequences grows exponentially. To overcome this barrier, we propose a novel algorithmic framework that consists of a specifically designed order (termed ABC order) and an algorithm that computes the minimized subsequence under an ABC order in polynomial time. We first show that the ABC order exhibits the desired property and the probability of hash collision using the ABC order is close to the Jaccard index. We then show that SubseqHash overwhelmingly outperforms the substring-based seeding methods in producing high-quality seed-matches for three critical applications: read mapping, sequence alignment, and overlap detection. SubseqHash presents a major algorithmic breakthrough for tackling the high error rates and we expect it to be widely adapted for long-reads analysis.
Availability: SubseqHash is freely available at `https://github.com/Shao-Group/subseqhash`.
Contact: mxs2589@psu.edu

1 Introduction

Transforming a sequence into a list of seeds (also known as markers) that are then processed in place of the original sequence is a common approach in sequence analysis as a remedy for the dreadfully expensive full-length comparisons. The resulting seeds of such a transformation are often regular in length and much shorter than the original sequences, making it possible to apply efficient data structures and fast algorithms. For example, in read mapping, the popular seed-and-extend strategy [2, 3] first identifies seed-matches (i.e., pairs of identical seeds; also known as anchors) between a read and a reference, then performs local alignment around these seeds to look for statistically significant matches. In sequence alignment, seed-matches across sequences are first identified, followed by efficient chaining algorithms to find a colinear chain of matching seeds that maximizes a scoring function [29, 1, 17]. In genome assembly, a (sparse) de Bruijn graph can be constructed in which seeds are used as vertices and two seeds are linked by an edge if they are adjacent in some reads [32, 14, 4, 23, 36]. To mitigate all-versus-all pairwise comparisons in applications such as aligning multiple sequences and constructing overlap/string graphs [19, 30, 9], seeds can be used to bucket sequences (i.e., assigning a sequence into buckets labeled by its own seeds), followed by pairwise comparisons in individual buckets [40, 5]. The efficiency and accuracy of these methods heavily rely on the quality of the generated seeds. Desired properties include high sensitivity (i.e., biologically related sequences producing many seed-matches) and a low false positive rate (i.e., unrelated sequences producing few seed-matches).

Arguably the most widely used seeds are simply kmers (i.e., substrings of fixed length k). Sketching approaches such as Minimizers [39, 35, 34, 27] or syncmers [13] are often combined to select a subset of kmers

aiming for scaling. Seeding methods using kmers have gained success in almost all aspects of sequence analysis especially on data with low error rates. However, requiring exact matches of k consecutive characters becomes less effective in comparing biologically related sequences with high mutation rates or error rates. Such scenarios include comparing homologous genes or whole genomes from distant species and processing long-read sequencing data generated by PacBio [33] and Oxford Nanopore [18] technologies. Observe that a single mutation/error can change k consecutive kmers in the sequence and the probability of a kmer remaining intact under a uniform mutation model decreases exponentially as k grows [6]. This greatly challenges the kmer-based seeding methods and puts them in a dilemma: choosing a large k results in few seed-matches even in biologically related sequences (i.e., low sensitivity), while making k too small suffers from a high false positive rate as unrelated sequences can, by chance, share many short common substrings. Existing tools such as KmerGenie [10] help select a size of kmers that balances its sensitivity and false positive rate based on the data. But due to the intrinsic weakness of kmers against mutations, even an optimal choice of k can still produce unsatisfactory results.

Alternative approaches have been proposed to collect nonconsecutive characters as seeds. Spaced seeds [8, 24] are extracted by applying a predefined pattern such as 1110111 on a string where a 1 means the character at that position is taken and a 0 means it is ignored/masked. The masked positions allow seed-matches to occur over substitutions, but because of the fixed pattern, spaced seeds can only handle substitutions at predefined locations and are still vulnerable to insertions and deletions (indels). Indel seeds [26] use patterns with wild-cards to accommodate certain numbers of indels. But again only indels at the predefined regions can be managed. Using multiple patterns [22, 20, 41, 15] alleviates the restrictions of a single pattern at the cost of more computations but still cannot handle mutations/errors at arbitrary locations.

It is also worth mentioning that there have been successful seeding methods that combine two or more pre-extracted (shorter) seeds. For example, grouped methods [11, 12] use two or more independently produced kmers as seeds. The Order Min Hash (OMH) approach [28] selects multiple kmers from a sequence with relative positions preserved. The recently proposed kmer-alternative method strobemer and its variants [37, 25, 38] pick and concatenate kmers from multiple consecutive predetermined windows. This provides more flexibility on the spacing between extracted kmers and therefore is less susceptible to different mutation rates. Nonetheless, since these methods all use kmers as building blocks, they cannot fully resolve the drawback of kmer-based methods. In addition, we consider these approaches orthogonal to the *basic* seeding methods (such as kmers and our method described below) in the sense that they can be applied on top of any kind of basic seeds.

In this paper, we explore the use of subsequences, rather than substrings, as seeds. The key observation is that two similar strings may share few or even zero substrings (of length k) but can contain many common subsequences (of length k or longer). Consider an example with two similar strings of length 7: $\boldsymbol{s}$ = ACGCCTA and $\boldsymbol{t}$ = ACGGCTA that differ by one substitution in the middle (i.e., their edit distance is 1). Clearly, $\boldsymbol{s}$ and $\boldsymbol{t}$ do not share any kmer for $k \geq 4$, leading to zero seed-matches for any kmer-based seeding methods (when $k \geq 4$ is used). On the other hand, two-thirds of the total 21 unique length-4 subsequences of $\boldsymbol{s}$ are also subsequences of $\boldsymbol{t}$. In fact, the Jaccard index between the two sets of length-4 subsequences of $\boldsymbol{s}$ and $\boldsymbol{t}$ is $14/(21 + 24 - 14) \approx 0.45$. According to the property of MinHash [7], if the *smallest* subsequences of length 4 (with respect to a fully random order, i.e., an order picked uniformly at random from all possible orders over strings of length 4) from $\boldsymbol{s}$ and $\boldsymbol{t}$ are picked as their respective seeds, then the probability of hash collision (i.e., producing a seed-match) is also about 0.45. Although rather simple, this example demonstrates the potential advantage of using subsequences as seeds: as it is more tolerant to edits happening at any position, it is more likely to produce seed-matches for similar sequences, and therefore provides enhanced sensitivity, particularly for data with a high mutation/error rate.

To this end, we present a subsequence-based seeding method named SubseqHash. It maps a string of length n to its smallest subsequence of length k (i.e., the seed), $k < n$, according to a given order over all length-k strings. There is one caveat: the number of subsequences grows exponentially in a string, which makes the computation of the smallest subsequence intractable when a fully random order is used. To overcome this difficulty, we propose a new algorithmic framework, consisting of a specifically designed order, named ABC order, and an algorithm that finds the smallest subsequence under an ABC order in polynomial time (Section 2). We show experimentally that an ABC order exhibits similar properties as a fully random order and that the probability of hash collision is close to that when a fully random order is used (Sections 3.1 and 3.2). We finally demonstrate the superiority of SubseqHash over the kmer-based seeding method Minimizer in several applications, including sequence alignment, read mapping, and overlap detection (Sections 3.4, 3.5, and 3.6).

2 SubseqHash

2.1 Definitions of SubseqHash

Let $\boldsymbol{x}$ be a string of length n over an alphabet Σ. Given an integer $k \leq n$, denote by $S_k(\boldsymbol{x})$ the set of all subsequences of $\boldsymbol{x}$ of length k. Let π be a permutation of Σ^k, i.e., π defines an order over all possible strings of length k. Define $h_\pi(\boldsymbol{x})$ as the smallest string in $S_k(\boldsymbol{x})$ according to π. In other words, the function h_π maps a string of length n to its smallest (defined by π) subsequence of length k; formally, $h_\pi(\boldsymbol{x}) = \arg\min_{\boldsymbol{z} \in S_k(\boldsymbol{x})} \pi(\boldsymbol{z})$, where $\pi(\boldsymbol{z})$ is the rank of $\boldsymbol{z}$ in the order defined by π. We use SubseqHash to term such a hashing function h_π.

2.2 Probability of Hash Collision

The intuition behind SubseqHash is that a few edits between two sequences may destroy most of their common substrings but many common subsequences can survive. We use the Jaccard index to measure the similarity of two sets. Given two strings $\boldsymbol{x}$ and $\boldsymbol{y}$, the Jaccard index for their subsequences of length k is defined as $J(\boldsymbol{x}, \boldsymbol{y}) := |S_k(\boldsymbol{x}) \cap S_k(\boldsymbol{y})|/|S_k(\boldsymbol{x}) \cup S_k(\boldsymbol{y})|$. The Jaccard index for their substrings can be defined similarly. In Supplementary Notes 1 and 2, we estimate and compare the Jaccard index for subsequences and substrings; in general, the Jaccard index for subsequences is larger than that for substrings, verifying the intuition.

We say an order π over Σ^k is *fully random* if π is drawn uniformly at random from all orders. For a fully random order π, the probability of hash collision of h_π is exactly the Jaccard index for subsequences according to the property of MinHash [7], i.e., $\Pr(h_\pi(\boldsymbol{x}) = h_\pi(\boldsymbol{y})) = J(\boldsymbol{x}, \boldsymbol{y})$ for any two strings $\boldsymbol{x}$ and $\boldsymbol{y}$. If π is not fully random then this may not hold. For example, when the lexicographic order is used, the empirical probability of hash collision reduces considerably (Section 3.2).

2.3 Algorithmic Framework for Constructing SubseqHash

The complexity of calculating $h_\pi(\boldsymbol{x})$ for a given string $\boldsymbol{x}$ also depends on the choice of π. For a fully random order π, one can compute $h_\pi(\boldsymbol{x})$ by enumerating all subsequences of $\boldsymbol{x}$ and picking the smallest one. Another approach is to traverse all strings of length k down the order π and return the first one that is a subsequence of $\boldsymbol{x}$. (Determining if a string $\boldsymbol{z}$ is a

subsequence of another string $\boldsymbol{x}$ can be done in $O(|\boldsymbol{z}| + |\boldsymbol{x}|)$ time, where $|\cdot|$ denotes the length of a string.) Both approaches run in exponential time. On the other hand, when the lexicographic order π is used, $h_\pi(\boldsymbol{x})$ can be computed in linear time; the downside is that the probability of hash collision gets reduced significantly as stated above.

We propose a novel approach to balance performance and efficiency. The idea is to use a special order π that allows for computing $h_\pi(\boldsymbol{x})$ in polynomial time. Such a special order is not fully random, but is designed to be "quite" random, and therefore achieves a probability of hash collision comparable with a fully random order. The special order π and the polynomial time algorithm are described in the next two sections.

2.4 The ABC Order

We start with designing an order π over Σ^k, named ABC order. Let $d \geq 1$ be an integer parameter. Essentially, π is given as a scoring function that maps a string $\boldsymbol{z} \in \Sigma^k$ to a pair $\pi(\boldsymbol{z}) := (\psi(\boldsymbol{z}), \omega(\boldsymbol{z}))$ where $\psi(\boldsymbol{z}) \in \{0, 1, \ldots, d-1\}$ and $\omega(\boldsymbol{z}) \in \mathbb{R}$. Then all strings in Σ^k are ordered as follows: for $\boldsymbol{z}, \boldsymbol{z'} \in \Sigma^k$, define $\pi(\boldsymbol{z}) < \pi(\boldsymbol{z'})$ if and only if $\psi(\boldsymbol{z}) < \psi(\boldsymbol{z'})$, or $\psi(\boldsymbol{z}) = \psi(\boldsymbol{z'})$ and $|\omega(\boldsymbol{z})| > |\omega(\boldsymbol{z'})|$. The construction of $\pi(\cdot)$ (see below) makes it extremely unlikely to have $\psi(\boldsymbol{z}) = \psi(\boldsymbol{z'})$ and $|\omega(\boldsymbol{z})| = |\omega(\boldsymbol{z'})|$ for $\boldsymbol{z} \neq \boldsymbol{z'}$. When such a rare case happens, we define $\pi(\boldsymbol{z}) < \pi(\boldsymbol{z'})$ if and only if $\boldsymbol{z}$ is lexicographically smaller than $\boldsymbol{z'}$.

We specify such a function π for DNA strings by assuming $\Sigma = \{\mathrm{A} = 1,\ \mathrm{C} = 2,\ \mathrm{G} = 3,\ \mathrm{T} = 4\}$. The function π is governed by three (random) tables A, B, and C; hence the name. Table A is a 3D real matrix of dimension $k \times d \times |\Sigma|$, i.e., $A \in \mathbb{R}^{k \times d \times |\Sigma|}$. Table B is also of dimension $k \times d \times |\Sigma|$, where $B[i][j][\sigma] \in \{(+1,+1), (+1,-1), (-1,+1), (-1,-1)\}$, $1 \leq i \leq k$, $0 \leq j \leq d-1$, and $\sigma \in \Sigma$. Table C has dimension $k \times |\Sigma|$, where $C[i][\sigma] \in \{0, 1, \cdots, d-1\}$, $1 \leq i \leq k$ and $\sigma \in \Sigma$.

These three tables are randomly generated in the following way. Each element $A[i][j][\sigma]$ is drawn independently and uniformly at random from a predetermined subset of $\mathbb{R}$; our implementation uses $\left[2^{30}, 2^{31}\right]$. For any fixed i and j, $1 \leq i \leq k$ and $0 \leq j \leq d-1$, $B[i][j][\sigma]$ is picked from the four pairs $\{(+1,+1), (+1,-1), (-1,+1), (-1,-1)\}$ uniformly at random without replacement, i.e., $B[i][j][\sigma_1] \neq B[i][j][\sigma_2]$ if $\sigma_1 \neq \sigma_2$. Last, for any fixed i, $1 \leq i \leq k$, each element $C[i][\sigma]$ is drawn independently and uniformly at random from $\{0, 1, 2, \ldots, d-1\}$. If the parameter $d \geq |\Sigma| = 4$, then we do this without replacement, i.e., $C[i][\sigma_1] \neq C[i][\sigma_2]$ if $\sigma_1 \neq \sigma_2$. Please see Supplementary Note 3 for an example.

Once the tables A, B, and C are generated, the scoring function $\pi(\boldsymbol{z}) = (\psi(\boldsymbol{z}), \omega(\boldsymbol{z}))$ is determined for any string $\boldsymbol{z} \in \Sigma^k$. Write $\boldsymbol{z} = z_1 z_2 \cdots z_k$, where $z_i \in \Sigma$, $1 \leq i \leq k$. For the sake of simplicity, denote $\psi(z_1 z_2 \cdots z_i)$ and $\omega(z_1 z_2 \cdots z_i)$ by ψ_i and ω_i, respectively; also denote the first and the second element in the pair $B[i][j][\sigma]$ by $B[i][j][\sigma]_1$ and $B[i][j][\sigma]_2$, respectively.

The initial values are set to $\psi_0 = 0$ and $\omega_0 = 0$. For $1 \leq i \leq k$, we use recurrences

$$\psi_i = (\psi_{i-1} + C[i][z_i]) \bmod d,$$

and

$$\omega_i = \omega_{i-1} \cdot B[i][\psi_i][z_i]_1 + A[i][\psi_i][z_i] \cdot B[i][\psi_i][z_i]_2.$$

Observe that if $d \geq 2$, then one edit in the string is guaranteed to alter the value of ψ, while two edits have a small chance (approximately $1/d$) to result in the same ψ. In addition, due to the use of -1 in table B, the value of ω can be substantially changed with even a single mutation. Combined, a few edits can cause a drastic change in both ψ and ω, and therefore the rank of strings in the order, which is a desired property; see Section 3.1 for more discussions.

2.5 Algorithm for Computing the Smallest Subsequence with an ABC Order

Let π be an ABC order. We design an efficient algorithm to find $h_\pi(\boldsymbol{x}) = \arg\min_{\boldsymbol{z} \in S_k(\boldsymbol{x})} \pi(\boldsymbol{z})$ for any given $\boldsymbol{x} = x_1 x_2 \cdots x_n \in \Sigma^n$. Since the scoring function $\pi(\boldsymbol{z})$ is defined by recurrences, it is natural to solve $h_\pi(\boldsymbol{x})$ using a dynamic programming algorithm. Consider a subproblem parameterized by l, i, and j, where $1 \leq l \leq n$, $1 \leq i \leq k$, and $0 \leq j < d$, which is defined to seek a subsequence $z_1 z_2 \cdots z_i$ of $x_1 x_2 \cdots x_l$ such that $\psi(z_1 z_2 \cdots z_i) = j$ and $\omega(z_1 z_2 \cdots z_i)$ is minimized or maximized. We calculate both the maximized and minimized values because they would be switched when encountering a pair from table B with -1 being its first element. Furthermore, both values are needed at the end since the definition of an ABC order requires finding the subsequence that maximizes the absolute value of $\omega(\cdot)$.

Formally, for each l, i, and j, we define subproblems:

$$T_{min}[l][i][j] := \min_{z \in S_i(x_1 x_2 \cdots x_l) \text{ and } \psi(z) = j} \omega(z)$$

and

$$T_{max}[l][i][j] := \max_{z \in S_i(x_1 x_2 \cdots x_l) \text{ and } \psi(z) = j} \omega(z)$$

They can be calculated with the recurrences below, in which $j' = (j - C[i][x_l] + d) \bmod d$.

$$T_{min}[l][i][j] = \min \begin{cases} T_{min}[l-1][i][j] \\ A[i][j][x_l] \cdot B[i][j][x_l]_2 + \begin{cases} +T_{min}[l-1][i-1][j'] \\ \quad (\text{if } B[i][j][x_l]_1 = +1) \\ -T_{max}[l-1][i-1][j'] \\ \quad (\text{if } B[i][j][x_l]_1 = -1) \end{cases} \end{cases}$$

$$T_{max}[l][i][j] = \max \begin{cases} T_{max}[l-1][i][j] \\ A[i][j][x_l] \cdot B[i][j][x_l]_2 + \begin{cases} +T_{max}[l-1][i-1][j'] \\ \quad (\text{if } B[i][j][x_l]_1 = +1) \\ -T_{min}[l-1][i-1][j'] \\ \quad (\text{if } B[i][j][x_l]_1 = -1) \end{cases} \end{cases}$$

Initially, for any $0 \leq l \leq n$, $T_{min}[l][0][0] = T_{max}[l][0][0] = 0$ and $T_{min}[l][0][j] = T_{max}[l][0][j] = \text{NaN}$ if $j \neq 0$. Tables T_{min} and T_{max} can then be filled using the above recurrences. Subsequently, for $0 \leq j < d$, we record the values

$$T[n][k][j] := \max\{|T_{min}[n][k][j]|, |T_{max}[n][k][j]|\}.$$

In these processes, if any of the three arithmetic operations $\{+, -, |\cdot|\}$ involves NaN as an operand, then the result is also an NaN. The $\min$ and $\max$ operations ignore NaN and only work on numerical operands, unless there is none, in which case an NaN is returned. At the end, we calculate

$$\psi(\boldsymbol{x}) = \min\{j \mid T[n][k][j] \neq \text{NaN}\} \text{ and } |\omega(\boldsymbol{x})| = T[n][k][\psi(\boldsymbol{x})].$$

The optimal subsequence, i.e., $h_\pi(\boldsymbol{x})$, can be obtained by traceback. The entire algorithm runs in $O(nkd)$ time.

2.6 Using SubseqHash in Practice

It is desirable for a seeding/hashing function to be *locality-sensitive*, i.e., the probability of hash collision for a pair of strings $\boldsymbol{x}$ and $\boldsymbol{y}$ is high if they are similar (say, measured with the edit distance), and at the same time such probability becomes low if $\boldsymbol{x}$ and $\boldsymbol{y}$ are not similar. These desired properties can also be interpreted as having high sensitivity (more true seed-matches) and a low false positive rate (fewer false seed-matches). For SubseqHash coupled with an ABC order, the choice of n and k balances these two measures. Generally speaking, a larger n lowers the false positive rate while a smaller n provides higher sensitivity; for a fixed n, increasing the difference between n and k improves sensitivity while decreasing the difference reduces the number of false seed-matches.

Again, two similar strings $\boldsymbol{x}$ and $\boldsymbol{y}$ are likely to share some (long) subsequences. In fact, assume the edit distance between two length-n strings $\boldsymbol{x}$ and $\boldsymbol{y}$ is e_1, then a shared subsequence of length k is guaranteed if $k \leq n - e_1$. In this case, the probability of hash collision under SubseqHash between $\boldsymbol{x}$ and $\boldsymbol{y}$ is strictly positive. However, the probability might be small so that one round of SubseqHash may not actually pick a common subsequence of the two strings (see Section 3.2 for some experimental results). To boost the chance of getting a seed-match, one can *repeat* SubseqHash several times independently, each of which uses a different set of random $A/B/C$ tables. Assume that p is the probability of hash collision of calling SubseqHash once, then with t repeats, the probability of having at least one seed-match is $1-(1-p)^t$. Repeats may also increase the false positive seed-matches, but it can be well controlled by picking a large n and a k that is close to n. Specifically, two dissimilar strings (i.e., their edit distance e_2 is large) of length n will not share any subsequence of length k if $k > n - e_2/2$. In Sections 3.4, 3.5, and 3.6, we show that repeats can boost sensitivity while maintaining a low false positive rate (i.e., high precision).

On the other hand, repeats are not as practical for substring-based seeding methods. This is because it is easy for two similar strings not to share any substring of a reasonable length. In fact, a shared substring of length k is guaranteed only if $k \leq n - ke_1$, as one edit can break up to k substrings of length k. When two (similar) strings do not share any length-k substring, a seed-match will not be produced regardless of the number of repeats. In the experiments, we include the comparison with "all-kmers" (i.e, every sliding window of length k in a sequence is collected as a seed), which is the limit of repeating Minimizers.

3 Results

3.1 Comparison of Orders

We propose a measure to characterize the similarity of neighboring strings in an order. Observe that, the neighboring strings in the lexicographic order are similar, while they are independent and therefore distant from each other in a fully random order. Let O be an order over Σ^k and let $O[i]$ be its i-th string. We define the $2w$ strings in a window centered at $O[i]$ as the neighboring strings of $O[i]$, where w is a parameter. We use $D_w(O,i) := \min_{i-w\leq j\leq i+w, j\neq i} edit(O[j], O[i])$ to quantify how similar $O[i]$ is with its neighboring strings, where $edit(\cdot,\cdot)$ denotes the edit distance. We finally calculate the *averaged minimum neighboring edit distance* (AMNED) over the m smallest (top-ranked) strings in the order, i.e., $D_{w,m}(O) := \sum_{i=1}^{m} D_w(O,i)/m$. We consider top-ranked strings in an order as they are more likely to be hashed to in SubseqHash.

We compare the AMNEDs of ABC orderings, fully random orders, and the lexicographic order for strings of length $k = 15$. To generate the top m strings in an ABC order, we randomly generate tables A, B, and C (with 3 choices of $d = 1, 11, 31$), calculate the score $(\psi(\cdot), \omega(\cdot))$ of all strings in Σ^k, sort them, and pick the top m strings. The top m strings of a fully random order is generated by independently simulating random strings of length k until m distinct ones are available. Note that the AMNED is 1 for the lexicographic order regardless the choice of w and m.

Fig. 1 and Supplementary Figure 3 reports the averaged AMNEDs and the standard deviation for the above three orders (10 repetitions for the ABC order and the fully random order) using different choices of w and m. The average AMNED for the ABC order is reasonably large, suggesting that the ABC order is "quite" random, in the sense that nearby strings are dissimilar. There is still a gap between an ABC order and a fully random order, but the gap is gradually decreased as m grows. Changing d from 1 to 11 for an ABC order significantly increases the AMNED, suggesting the effectiveness of using table C. There is a small growth when d is further increased to 31; we therefore pick $d = 11$ in the experimental studies.

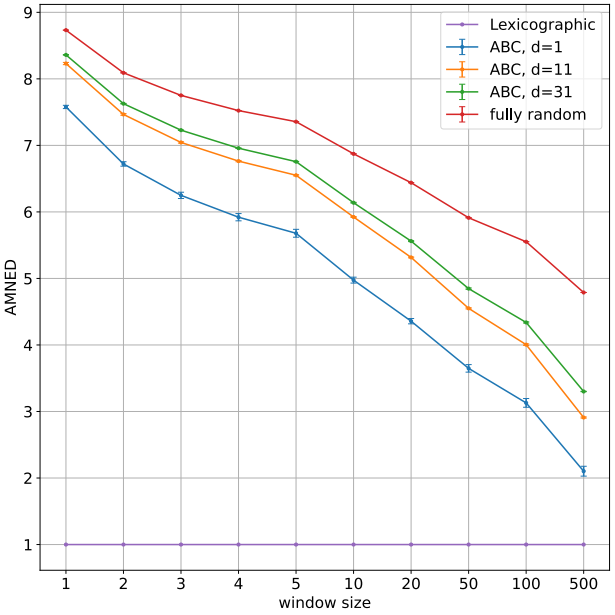

Fig. 1: The AMNED of different orders over strings of length $k = 15$ evaluated with varying w (x-axis) and $m = 10,000$. The point and error bar show the mean and standard deviation over 10 individual runs. Results for different values of m are available in Supplementary Figure 3.

3.2 Comparison of Probability of Hash Collision

We compare the probability of hash collision achieved by Minimizer and SubseqHash.Each seeding method takes a pair of strings $(\boldsymbol{x}, \boldsymbol{y})$ as input, and extracts a single seed from each string. For Minimizer, the seed of $\boldsymbol{x}$ is the smallest kmer among the $(|\boldsymbol{x}| - k + 1)$ kmers in $\boldsymbol{x}$ according to a fully random order. For SubseqHash, the seed of $\boldsymbol{x}$ is the smallest subsequence of length k in $\boldsymbol{x}$, according to the chosen order. For the lexicographic order, linear algorithm exists to find the optimal seed; for the ABC order, algorithm in Section 2.5 is used; for a fully random order we use a brute-force approach to find the smallest seed (and hence we are not able to report the results for large k in Supplementary Figures 5). A seed of $\boldsymbol{y}$ will be extracted independently but with the shared order used for $\boldsymbol{x}$. We then check if the two seeds are identical (i.e., a hash collision).

We use simulations to estimate the probability of hash collisions. To simulate pairs of strings, we start with $\boldsymbol{x}$ being a random string of length n, $n = 20$ or $n = 30$. We then apply n random evolutionary events sequentially on $\boldsymbol{x}$, with each event with probability of $1/3$ being a

substitution, a deletion, or an insertion. We make sure that each position can be only mutated once. We collect both the intermediate $n-1$ strings and the final string, resulting in n pairs of strings $(\boldsymbol{x}, \boldsymbol{y_i}), i = 1, 2, \cdots, n$. All pairs are categorized according to the edit distance, i.e., pair $(\boldsymbol{x}, \boldsymbol{y_i})$ is put into the j-th category if $edit(\boldsymbol{x}, \boldsymbol{y_i}) = j$. Note that there are i mutations simulated from $\boldsymbol{x}$ to $\boldsymbol{y_i}$, but it is not necessarily true that $edit(\boldsymbol{x}, \boldsymbol{y_i}) = i$. Notice also that it is possible that $|\boldsymbol{y_i}| \neq |\boldsymbol{x}|$, but a seed of the same length (a kmer for Minimizer and a subsequence of length k for SubseqHash) will be extracted from them.

We simulate 10,000 pairs of strings following above procedure, and in each of the 10 categories where edit distance is from 1 to 10, we calculate the frequency of hash collisions and use it as an estimation of the probability of hash collision. The results are shown in Fig. 2 and Supplementary Figures 4 and 5. Minimizer and SubseqHash are compared when extracting seeds of the same length (the same k). Observe that in all settings SubseqHash coupled with ABC order achieves much higher probability than Minimizer when the edit distance is in a range of 1 to 5, indicating the superiority of SubseqHash over Minimizer in hashing similar strings but with high error rates. The probability of hash collision of SubseqHash coupled with the lexicographic order is very similar to that of the Minimizer, suggesting the necessity of a more random order (than lexicographic order) to make SubseqHash more sensitive. The probability of hash collision of SubseqHash gets much improved when $d = 11$ is used in ABC order than $d = 1$, again indicating the effectiveness of table C. The performance of SubseqHash with fully random orders gives the highest probability (i.e., the Jaccard index) among possible orders, but the curves from ABC orders when $d = 11$ and $d = 31$ are very close to them, indicating that the ABC order (with large d) is nearly optimal.

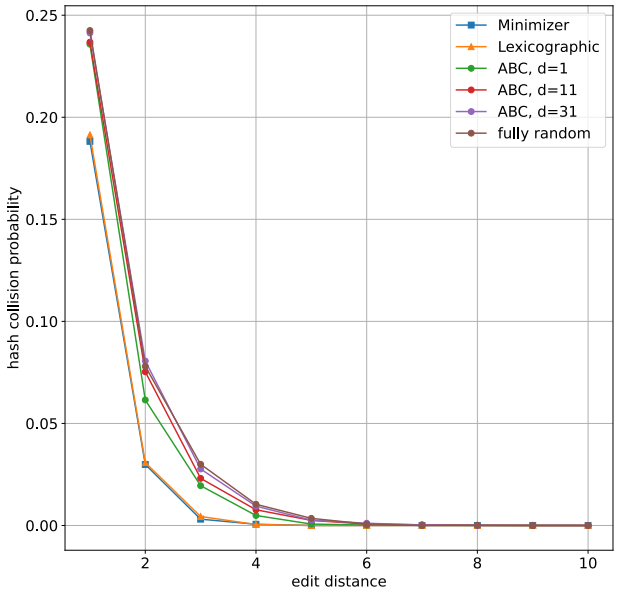

Fig. 2: The probability of hash collision estimated, using simulations, for different seeding methods with $n = 20$ and $k = 16$. More results with different n and k are available in Supplementary Figures 4 and 5.

3.3 Evaluating Seeding Methods

We discuss appropriate measures to evaluate seeding methods for tasks involving sequence comparison, such as sequence alignment, read mapping, and overlapping read detection. Despite variations in methods for these tasks, they all follow a two-step procedure, consisting of a *seeding step* and a *post-seeding step*. The seeding step treats the sequences/reads independently and typically applies a seeding method to sliding windows of a sequence/read, resulting in a list of seeds. Measures for this step include the running time of the seeding method, as well as the density of the seeds, defined as the number of seeds produced from a sequence divided by the length of the sequence.

The post-seeding step uses the generated seeds to compare sequences. We emphasize that in this step only the matched seeds between compared sequences are used, while unmatched seeds are discarded. For example, in the co-linear chaining approach for sequence alignment, the set of seed-matches serves as the input of the chaining algorithm. In the seed-and-extend scheme for mapping reads, each individual seed-match will be examined in the extension. When detecting overlapping reads, a pair of reads will be determined as "candidate" (which will be then subject to more fine-grained procedure such as chaining to decide overlapping) if there exists one (or more) seed-matches. Therefore, it is the quantity and quality of seed-matches, rather than that of seeds, that determine the running time of the post-seeding step and the accuracy of the outcomes.

In Sections 3.4 and 3.5, we report the density and running time of different seeding methods, as well as the quantity and quality of the resulting seed-matches to evaluate their impacts on the post-seeding step. In summary, SubseqHash runs much slower than substring-based methods such as Minimizers. When repetitions are applied to SubseqHash (see Section 2.6), it generates a much larger number of seeds than Minimizers, requiring more memory to store the seeds. However, SubseqHash outperforms other methods in generating high-quality seed-matches, thereby improving the accuracy of the final outcomes. See below for detailed analysis.

3.4 Application: Pairwise Sequence Alignment

We use simulations to test the performance of different methods with varying error rates. We simulate 100 pairs of sequences and report the average measures (described below). The first sequence in a pair is a random sequence of length $L = 100,000$; the second sequence is obtained by applying an edit, with probability of r equally distributed to insertion, deletion, and substitution, independently on every position of the first sequence, where r is a parameter specifying the error rate. The ground-truth alignment is saved for evaluation (see below).

In Fig.3, we present the density of SubseqHash and Minimizers using the simulated data described above. As expected, for any fixed window size n, the density of both methods decreases as the seed length k decreases, but the density of Minimizers decreases much more rapidly than that of SubseqHash. It should be noted that when repetitions are applied to SubseqHash, the density and total number of seeds should be multiplied by the number of repetitions. Furthermore, we provide a comparison of the running time of different seeding methods in Supplementary Tables 6, 7, 8, 9. Consistent with the theoretical analysis, SubseqHash typically runs 24 to 270 times slower than Minimizers.

We now assess the quality of resulting seed-matches. In either Minimizer or SubseqHash, a seed-match specifies an alignment among k identical characters. We define a seed-match to be *true* if at least 50% of the k aligned characters appear in the ground-truth; otherwise it is considered to be a *false* seed-match. See Fig. 4 for an example. We do not require all k aligned characters to agree with the ground-truth to be considered as a true seed-match as the ground-truth alignment may not reflect the most parsimonious alignment especially when the mutation rate is high. Nevertheless, 50% matched characters certainly indicate that the locations of two seeds are anchored correctly which is adequate for downstream use.

In Fig. 5 and Supplementary Figure 7, we present the relationship between the number of seed-matches and the ratio of true seed-match ratio (defined

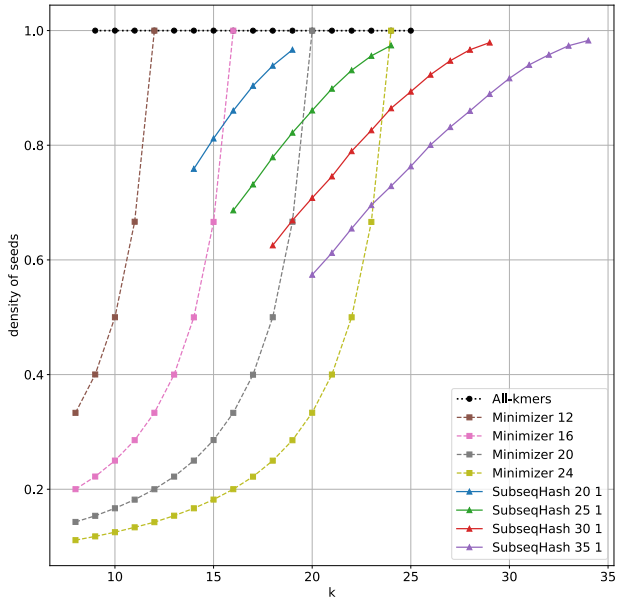

Fig. 3: The density of different seeding methods on simulated sequences. In the legend, "all-kmers" means every single kmer is collected as seed, $k = 9, 10, \cdots, 25$. "Minimizer n" means a window size of n; with increasing density, the list of k used for each line is $k = 8, 9, \cdots, n-1$. "SubseqHash n t" means a window size of n and repeating t times. For $n = 20$, k is from 14 to 19; for $n = 25$, k is from 16 to 24; for $n = 30$, k is from 18 to 29; for $n = 35$, k is from 20 to 34. For all methods, points with varying k but the same n are connected by lines. These parameters are also used in all experimental studies of Sections 3.4 and 3.5.

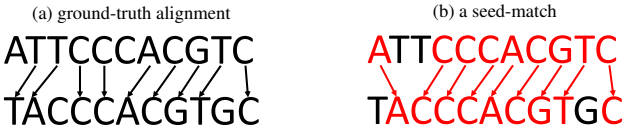

Fig. 4: (a) The ground-truth alignment between two sequences; note that this alignment is not the most parsimonous (i.e., minimizing edit distance) alignment. (b) A seed-match where the identical seed is *ACCCACGTC*. Among the nine matched chararcters, five of them (55.6%) exist in the ground-truth alignment. Hence this seed-match is a *true* one.

as the number of true seed-matches divided by the number of seed-matches). As explained in Section 3.3, the number of seed-matches has a strong correlation with the execution time of the post-seeding step, while the true seed-match ratio reflects the accuracy of seed-matches. By comparing the y-coordinates of different methods at a fixed x-coordinate, one can assess their accuracy at the same level of running time. SubseqHash without repetitions demonstrates a similar performance to Minimizers in the range of small numbers of seed-matches. SubseqHash with repetitions achieves a much higher true ratio than all-kmers when the number of seed-matches falls between approximately 10,000 and 30,000. Beyond this range, SubseqHash and other methods are not comparable.

It is more desirable for a seeding method to generate true seed-matches that span a larger range of the sequence, rather than ones clustered together [37]. We say a character in a sequence is *covered* by a seed-match if it is one of its k aligned characters. The *coverage of true seed-matches* (true coverage for short) is the percentage of characters in both sequences that are covered by at least one true seed-match; the *coverage of false seed-matches* (false

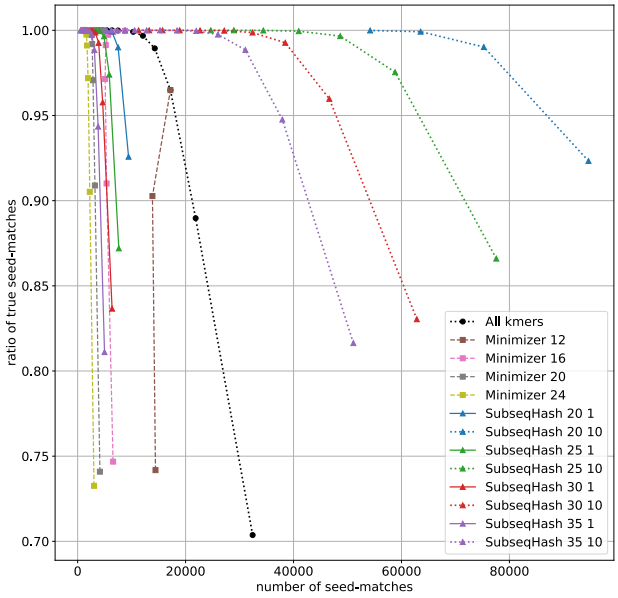

Fig. 5: The number of seed-matches and the true seed-matches ratios on simulated sequences with error rate = 15%. Figure is cropped to only show the portion with high ratio ($\geq$ 70%); complete results are shown in Supplementary Tables 2, 3, 4, 5. Results for error rates 5%, 10%, and 20% are available in Supplementary Figure 7.

coverage for short) is defined in the same way but counting false seed-matches. Higher true coverage reflects higher sensitivity, and can facilitate downstream chaining procedure to produce more accurate and faster sequence alignment. Lower false coverage reduces the likelihood of producing incorrect alignments. We report the average true/false coverages of different methods in Fig. 6 and Supplementary Figure 8. A single run of SubseqHash outperforms (i.e., higher true coverage at the same false coverage) Minimizer and all-kmers when error rate is 5% or 10%, and achieves similar performance at 15% and 20% error rates. SubseqHash

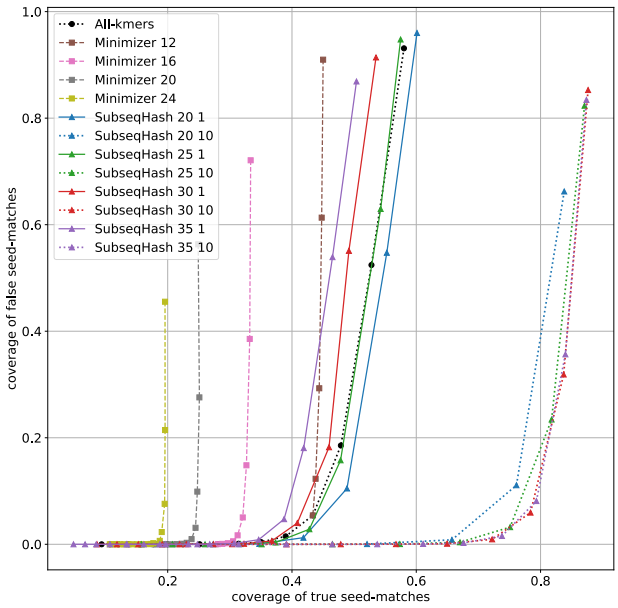

Fig. 6: The coverages of true and false seed-matches for different seeding methods on simulated sequences with error rates = 15%. Results for different error rates are available in Supplementary Figure 8.

with 10 repetitions outperforms others by a large margin at all error rates: specifically, the highest true coverage achieved by Minimizer/all-kmers at a false coverage lower than 5% are 85.1%, 61.4%, 38.9%, and 23.1%, respectively, for the 4 error rates, while the numbers for SubseqHash are 98.0%, 90.8%, 75.3%, and 55.0%, respectively.

3.5 Application: Read Mapping

We then compare SubseqHash with other seeding methods on mapping reads to the reference genome. We utilize three real PacBio datasets from [5] on *E. coli* (SRX533603), *S. cerevisiae* (SRX533604), *D. melanogaster* (SRX499318). To construct a ground-truth for evaluation, we align the reads to the corresponding reference genomes using minimap2 [21] with the default parameters (`-cx map-pb`). For reads that have a mapped region of at least 2,000 bp and a mapping quality of at least 10, we trim the read to only keep the mapped portion. Reads with multiple qualified mappings are discarded. This produces the input reads and their ground-truth alignments. To eliminate the potential biases in the ground-truth created by minimap2, which internally uses Minimizers for seeding, we include a simulated dataset obtained with PBSIM2 [31] using the same statistics as the PacBio *D. melanogaster* dataset. A total of 287,648 reads are simulated from the X chromosome, with ground-truth alignment saved from simulation. We randomly sample 1000 reads from each of the 4 datasets for this experiment. The same set of seeding methods (Minimizer, all-kmers, and SubseqHash) is applied to generate seeds for both reads and the reference genomes. The density of different methods are illustrated in Supplementary Figures 9, 10, which show very similar results with Fig. 3.

To assess the quality of seed-matches, the same definitions of true and false seed-match used for pairwise sequence alignment (Section 3.4) is also used in this experiment. When calculating the true/false coverages, only the covered characters on reads are considered (instead of on both the reads and the reference genomes). Fig. 7 and Supplementary Figure 11 show the number of seed-matches and the ratio of true seed-matches for different methods averaged over the 1000 reads. At the same level of seed-matches,

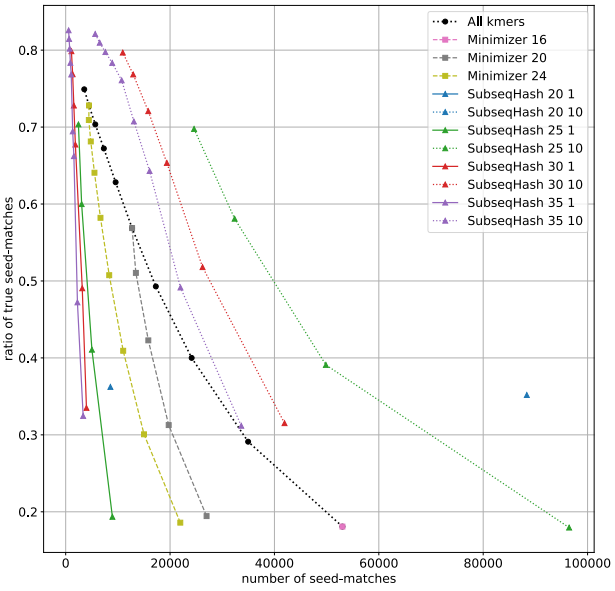

Fig. 7: The number of seed-matches and the ratio of true seed-matches for different seeding methods evaluated on *D. melanogaster* SRX499318 dataset. Figure is cropped to show high ratios ($\geq 15\%$); complete results are given in Supplementary Tables 10, 11, 12, 13. Results for other datasets are shown in Supplementary Figure 11.

SubseqHash with 10 repetitions can achieve much higher ratio of true seed-matches than all-kmers and Minimizers, verifying the effectiveness of repetitions for sequence mapping. Fig. 8 and Supplementary Figure 12 compares the true/false coverages of different methods. SubseqHash (without repeating) can obtain higher true coverage on the same foot of false coverage than all-kmers and Minimizers. Again SubseqHash with repeating 10 times outperforms all others substantially. To give some concrete numbers, the highest true coverage achieved by Minimizer/all-kmers at a false coverage lower than 10% are 38.2%, 39.0%, 31.4%, and 34.2%, respectively, for the 4 datasets, while the numbers for SubseqHash are 74.5%, 67.3%, 52.5%, and 57.5%, respectively.

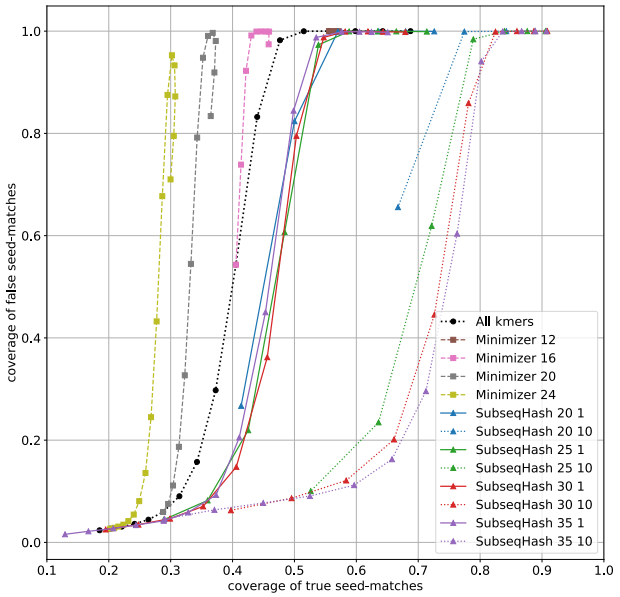

Fig. 8: The coverages of true and false seed-matches for different seeding methods on *D. melanogaster* SRX499318 dataset. Results for other datasets are available in Supplementary Figure 12.

3.6 Application: Overlap Detection

State-of-the-art methods for genome assembly using long-reads sequencing data often rely on an accurate overlap graph, in which vertices are reads and overlapping reads are connected with edges. A straightforward approach for constructing the overlap graph given a set of sequences (i.e., long reads) is performing all-versus-all comparisons, but it certainly does not scale. Seeding methods can be used to detect overlapping pairs while being able to scale. More specifically, a seeding method first transforms each sequence into seeds, then reports pairs of sequences that have at least one seed-match as candidate overlapping pairs. This is certainly a coarse model as for a real overlap detection tool, one would usually perform multiple steps of seed-preprocessing such as subsampling and filtering; then in the detection phase (post-seeding step), different thresholds for the number of seed-matches and location information of the seeds can be used; lastly, the candidate overlapping pairs are often verified with a fine-grained comparison (e.g., a local alignment) before the final output. All these steps make the overlap results more accurate, but because they can be applied regardless of the seeding methods used, we opt to omit them in this experiment so we can focus on a direct comparison of different seeding methods.

We use the same 4 datasets in Section 3.5 in this experiment. We sample 10,000 reads from each dataset; a pair of reads are considered truly

overlapping (i.e., ground-truth) if their mapped regions on the reference genome overlap by at least 15bp. To measure the candidate overlapping pairs reported by a seeding method, we define sensitivity as the fraction of ground-truth pairs that are identified by a seeding method; define precision as the fraction of all reported pairs that are correct according to the ground-truth.

The precision-sensitivity curves for different methods are shown in Fig. 9 and Supplementary Figure 13. Comparing with seeding for sequence alignment and read mapping, here we use a larger window size mainly to reduce false pairs. For each window size n used in Minimizers, six seed lengths k evenly spaced between 10 and n are included (when $k = n$, all kmers are picked as seeds). For each n used in SubseqHash, the seed lengths $k = \lfloor 0.65n \rfloor$, $\lfloor 0.7n \rfloor$, $\lfloor 0.75n \rfloor$, $\lfloor 0.8n \rfloor$, $\lfloor 0.85n \rfloor$ are tested. We include the results of SubseqHash without repetition and of 10 repeats for each of the parameters above. When repetitions apply, the overlapping pairs are simply the union of all 10 runs. Observe that on all 4 datasets, SubseqHash without repetition already shows better accuracy (i.e., higher precision at the same sensitivity level) than Minimizers and all-kmers with a few exceptions at sensitivity near 1.0. In this region, both methods suffer from extremely low precision which indicates that refining steps are necessary and raw seeds comparison at sensitivity close to 1.0 may not be truly informative. With 10 repetitions, the sensitivity of SubseqHash is significantly boosted while outperforming Minimizers and all-kmers substantially. For example, when the sensitivity is set to be at least 80%, the highest precisions achieved by Minimizer/all kmers for the 4 datasets are 62.9%, 38.9%, 8.7%, and 11.7%, while the numbers for SubseqHash are 85.9%, 78.0%, 23.9%, and 41.7%, respectively.

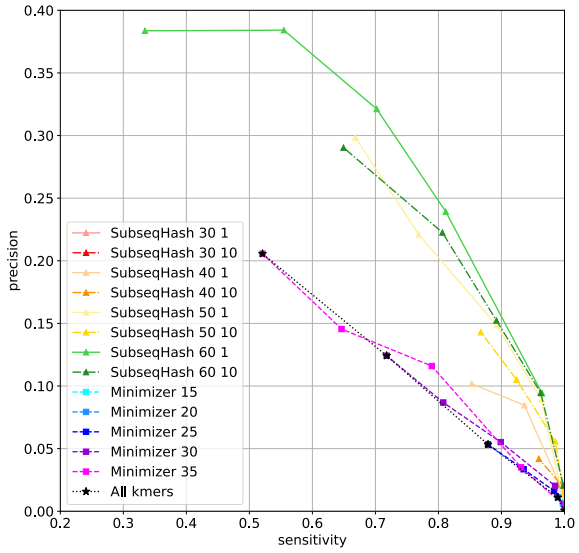

Fig. 9: Overlap detection results on reads sampled from *D. melanogaster* SRX499318 dataset. Results for other datasets are available in Supplementary Figure 13.

4 Discussion

We investigated SubseqHash, a new approach that uses the minimized subsequence as seed. We figured that the probability of hash collision is determined by the shared order. We therefore studied this core algorithmic formulation: seek an order π over all strings of length k such that π is "as random as possible" and that an efficient algorithm that finds the smallest subsequence (according to π) in a string (of length n) can be designed. We gave a practical solution for this formulation, consisting of the so-called ABC order together with a dynamic programming algorithm runs in $O(nkd)$ time to find the minimized subsequence under an ABC order (where $d \geq 1$ can be picked by users). We demonstrated that nearby strings in an ABC order are distant from each other, a property exhibited in a fully random order, and that the probability of hash collision with an ABC order is close to the Jaccard index, achievable when a fully random order is used.

The superiority of SubseqHash over substring-based methods is in three folds. First, SubseqHash tolerates errors while substring-based methods require exact matches. Second, the probability of hash collision of SubseqHash is higher than that of Minimizer (when extracting seeds of the same length). Third, the performance of SubseqHash can be substantially boosted through repetition while for Minimizer this is not as practical. These merits make SubseqHash a more suitable choice for seeding sequencing data with high error/mutation rates.

We showed that SubseqHash coupled with the ABC order substantially outperformed Minimizer in generating high-quality seed-matches for three applications, sequence alignment, read mapping, and overlap detection. We emphasize that these experiments were designed for a direct comparison between different seeding methods, and therefore the evaluations were conducted at the level of seeds, rather than evaluating the eventual outcomes (e.g., alignments or the overlap graph). As seeding is a key step involved in these applications, we expect SubseqHash will be widely adapted and incorporated to improve their accuracies on the analysis of third-generation sequencing data.

Our algorithm to find a single seed takes $O(nkd)$ time, which is much slower than Minimizer that takes amortized $O(1)$ time to find a seed in a window. We note that the seeding step usually takes less time than the post-seeding step especially in applications that require all-pairs comparisons, as seeding scales linearly with respect to the number of sequences. Users may choose a smaller d, say $d = 4$ or even $d = 1$, to gain a speedup at the cost of slightly decrease of sensitivity. Interesting future directions include accelerating the current algorithm using techniques such as $A*$ heuristic searching [16] and parallel algorithms that have been successfully used to speed up dynamic programming algorithms. We also believe the core algorithmic formulation (stated in the first paragraph of this Section), which we find fascinating, can be further improved in achieving faster algorithm and/or higher probability of hash collision.

Acknowledgments

This work is supported by the US National Science Foundation (2019797 and 2145171 to M.S.) and by the US National Institutes of Health (R01HG011065 to M.S.). We thank Paul Medvedev for helpful discussions.

References

[1]Mohamed Ibrahim Abouelhoda and Enno Ohlebusch. Chaining algorithms for multiple genome comparison. *Journal of Discrete Algorithms*, 3(2-4):321–341, 2005.

[2]Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[3]Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[4]Anton Bankevich, Andrey V Bzikadze, Mikhail Kolmogorov, Dmitry Antipov, and Pavel A Pevzner. Multiplex de bruijn graphs enable genome assembly from long, high-fidelity reads. *Nature Biotechnology*, pages 1–7, 2022.

[5]Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature Biotechnology*, 33(6):623–630, 2015.

[6]Antonio Blanca, Robert S Harris, David Koslicki, and Paul Medvedev. The statistics of k-mers from a sequence undergoing a simple mutation process without spurious matches. *Journal of Computational Biology*, 29(2):155–168, 2022.

[7]Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, 1997.

[8]Andrea Califano and Isidore Rigoutsos. FLASH: A fast look-up algorithm for string homology. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'93)*, pages 353–359. IEEE, 1993.

[9]Haoyu Cheng, Gregory T Concepcion, Xiaowen Feng, Haowen Zhang, and Heng Li. Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. *Nature Methods*, 18(2):170–175, 2021.

[10]Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, 2014.

[11]Chen-Shan Chin and Asif Khalak. Human genome assembly in 100 minutes. *bioRxiv*, page 705616, 2019.

[12]Nan Du, Jiao Chen, and Yanni Sun. Improving the sensitivity of long read overlap detection using grouped short k-mer matches. *BMC Genomics*, 20(2):49–62, 2019.

[13]Robert Edgar. Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences. *PeerJ*, 9:e10805, 2021.

[14]Barış Ekim, Bonnie Berger, and Rayan Chikhi. Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. *Cell Systems*, 12(10):958–968, 2021.

[15]Martin Farach-Colton, Gad M Landau, Cenk S Sahinalp, and Dekel Tsur. Optimal spaced seeds for faster approximate string matching. *Journal of Computer and System Sciences*, 73(7):1035–1044, 2007.

[16]Pesho Ivanov, Benjamin Bichsel, and Martin Vechev. Fast and optimal sequence-to-graph alignment guided by seeds. In *Proceedings of the 26th International Conference on Research in Computational Molecular Biology (RECOMB'22)*, pages 306–325. Springer, 2022.

[17]Chirag Jain, Daniel Gibney, and Sharma V Thankachan. Co-linear chaining with overlaps and gap costs. In *Proceedings of the 26th International Conference on Research in Computational Molecular Biology (RECOMB'22)*, pages 246–262. Springer, 2022.

[18]Miten Jain, Sergey Koren, Karen H Miga, Josh Quick, Arthur C Rand, Thomas A Sasani, John R Tyson, Andrew D Beggs, Alexander T Dilthey, Ian T Fiddes, et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*, 36(4):338–345, 2018.

[19]Sergey Koren, Brian P Walenz, Konstantin Berlin, Jason R Miller, Nicholas H Bergman, and Adam M Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research*, 27(5):722–736, 2017.

[20]Gregory Kucherov, Laurent Noé, and Mikhail Roytberg. Multiseed lossless filtration. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(1):51–61, 2005.

[21]Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[22]Ming Li, Bin Ma, Derek Kisman, and John Tromp. PatternHunter II: Highly sensitive and fast homology search. *Journal of Bioinformatics and Computational Biology*, 2(03):417–439, 2004.

[23]Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, Max W Shen, Mark Chaisson, and Pavel A Pevzner. Assembly of long error-prone reads using de bruijn graphs. *Proceedings of the National Academy of Sciences*, 113(52):E8396–E8405, 2016.

[24]Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[25]Benjamin Dominik Maier and Kristoffer Sahlin. Entropy predicts fuzzy-seed sensitivity. *bioRxiv*, page 2022.10.13.512198, 2022.

[26]Denise Mak, Yevgeniy Gelfand, and Gary Benson. Indel seeds for homology search. *Bioinformatics*, 22(14):e341–e349, 07 2006.

[27]Guillaume Marçais, Dan DeBlasio, and Carl Kingsford. Asymptotically optimal minimizers schemes. *Bioinformatics*, 34(13):i13–i22, 2018.

[28]Guillaume Marçais, Dan DeBlasio, Prashant Pandey, and Carl Kingsford. Locality-sensitive hashing for the edit distance. *Bioinformatics*, 35(14):i127–i135, 2019.

[29]Gene Myers and Webb Miller. Chaining multiple-alignment fragments in sub-quadratic time. In *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms (SODA'95)*, volume 95, pages 38–47, 1995.

[30]Sergey Nurk, Brian P Walenz, Arang Rhie, Mitchell R Vollger, Glennis A Logsdon, Robert Grothe, Karen H Miga, Evan E Eichler, Adam M Phillippy, and Sergey Koren. HiCanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads. *Genome Research*, 30(9):1291–1305, 2020.

[31]Yukiteru Ono, Kiyoshi Asai, and Michiaki Hamada. PBSIM2: a simulator for long-read sequencers with a novel generative model of quality scores. *Bioinformatics*, 37(5):589–595, 09 2020.

[32]Mikko Rautiainen and Tobias Marschall. MBG: Minimizer-based sparse de Bruijn graph construction. *Bioinformatics*, 37(16):2476–2478, 2021.

[33]Anthony Rhoads and Kin Fai Au. PacBio sequencing and its applications. *Genomics, Proteomics & Bioinformatics*, 13(5):278–289, 2015.

[34]Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[35]Michael Roberts, Brian R Hunt, James A Yorke, Randall A Bolanos, and Arthur L Delcher. A preprocessor for shotgun assembly of large genomes. *Journal of Computational Biology*, 11(4):734–752, 2004.

[36]Jue Ruan and Heng Li. Fast and accurate long-read assembly with wtdbg2. *Nature Methods*, 17(2):155–158, 2020.

[37]Kristoffer Sahlin. Effective sequence similarity detection with strobemers. *Genome Research*, 31(11):2080–2094, 2021.

[38]Kristoffer Sahlin. Strobealign: flexible seed size enables ultra-fast and accurate read alignment. *Genome Biology*, 23(1):1–27, 2022.

[39]Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD/PODS'03)*, pages 76–85, 2003.

[40]Yan Song, Haixu Tang, Haoyu Zhang, and Qin Zhang. Overlap detection on long, error-prone sequencing reads via smooth q-gram. *Bioinformatics*, 36(19):4838–4845, 2020.

[41]Yanni Sun and Jeremy Buhler. Designing multiple simultaneous seeds for DNA similarity search. *Journal of Computational Biology*, 12(6):847–861, 2005.