A performant bridge between fixed-size and variable-size seeding

Arne Kutzner[1] Pok-Son Kim[2] and Markus Schmidt[1,*]

[1] Department of Information Systems, College of Computer Science, Hanyang University, 222 Wangsimni-ro, Seongdong-gu, Seoul, 04763, Republic of Korea
[2] Department of Mathematics, College of Science and Technology, Kookmin University, 77, Jeongneung-ro, Seongbuk-gu, Seoul, 02707, Republic of Korea

* To whom correspondence should be addressed. Email: schmidtm@hanyang.ac.kr

ABSTRACT

Seeding is usually the initial step of high-throughput sequence aligners. Two popular seeding strategies are fixed-size seeding (k-mers, minimizers) and variable-size seeding (MEMs, SMEMs, max. spanning seeds). The former strategy benefits from fast index building and fast seed computation, while the latter one benefits from high seed entropy. Here we build a performant bridge between both strategies and show that neither of them is of theoretical superiority. We propose an algorithmic approach for computing MEMs out of k-mers or minimizers. Further, we describe techniques for extracting SMEMs or maximally spanning seeds out of MEMs. A comprehensive benchmarking shows the practical value of the proposed approaches. In this context, we report about the effects and the fine-tuning of occurrence filters for the different seeding strategies.

KEYWORDS

high-throughput sequence alignment, minimizer, SMEM, FMD-Index, seed entropy.

INTRODUCTION

Most high-throughput read aligners [1-5] perform the following three steps: seeding [6, 7], seed processing (e.g. chaining, SoC) [8, 9] and dynamic programming [10, 11]. There are two techniques for seed computation: fixed-sized seeding [12] and variable-size seeding [13, 14]. Fixed-size seeding is usually done via k-mers or via their space efficient variant, minimizers [3]. Variable-size seeding, in turn, relies on some form of full-text search index as e.g. the FMD-index [13, 14]. Fixed-size seeding benefits from short runtimes for index construction and seed computation, while variable-size seeding benefits from the high entropy of the generated seeds [2, 3]. Here, we present an efficient algorithmic bridge for computing variable-size seeds out of fixed-size seeds. Hence, the performant behavior of fixed-size seeds becomes available with variable-size seeds as well.

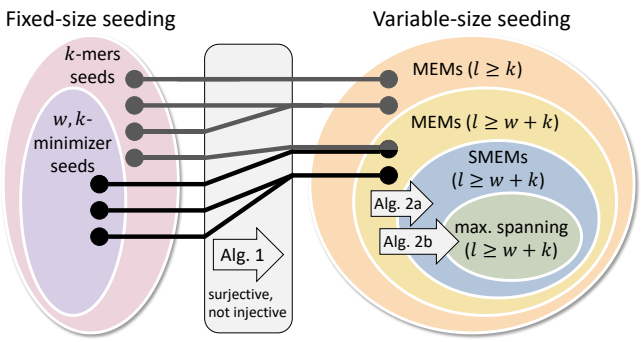

Figure 1. The drawing shows the hierarchy among several different seeding techniques for fixed-size seeding (k-mers and minimizers) and variable-size seeding (MEMs, SMEMs, max. spanning seeds). The ovals shall indicate subset relationships among the different seed sets. w is the window size for minimizers and k is the size of minimizers and k-mers. l is the length of variable-size seeds. Alg. 1 constructs MEMs of size $\geq k$ and $\geq w+k$ out of k-mers and w,k-minimizers, respectively. If the query interval of a MEM is not encompassed by another MEM, then it is a SMEM (super-maximal exact match). Maximally spanning seeds are MEMs that comprise at least one query position, where they are not covered by any longer MEM. Because the condition fulfilled by max. spanning seeds represents a strengthened variant of the condition for SMEMs, all max. spanning seeds are SMEMs. Alg. 2a in the methods section extracts SMEMs out of MEMs. Alg. 2b computes max. spanning seeds from MEMs (or SMEMs).

Fig. 1 shows a hierarchy among several different seeding techniques. Here, a k-mer is a seed represented as a triple (q,r,l) that expresses the equivalence of the query interval $[q,q+l)$ and the reference interval $[r,r+l)$, where $l=k$. w,k-minimizers represent a subset of k-mers. Via the algorithmic bridge proposed here (Alg. 1), it is possible to define a mapping between k-mers and MEMs (maximal exact matches, see [6]). The mapping is surjective, because every MEM has at least one corresponding k-mer. However, it is not injective, because the mapping of different k-mers can lead to the same MEM. The inverse mapping is achieved by simply computing all k-sized sub seeds of each MEM. Minimizers correspond to a subgroup within the MEMs due to size limitations. There are two further subgroups within MEMs: SMEMs (super-maximal exact matches [15]) and maximally spanning seeds [2]. In the results section, we show that SMEMs have a higher seed entropy than MEMs and that maximally spanning seeds have the highest seed entropy among all three. Here we understand entropy as the number of nucleotides that a single seed contributes to an accurate alignment on average. SMEMs can be performantly extracted from MEMs by Alg. 2a, which is defined in the methods section. For slightly higher computational costs, maximally spanning seeds can be computed out of MEMs by Alg. 2b.

An aligner can only find the correct location of a read, if there are seeds for this location. Therefore, the seeding defines an upper bound for the maximal accuracy of an aligner. The mapping between k-mers and MEMs (see Fig. 1) shows an equivalence of fixed-sized seeding and variable-size seeding with respect to such upper bound considerations. It can get disturbed by occurrence filtering techniques as they are practically used during seeding (e.g. occurrence filter in minimap2 [3] or MA [2]). However, by fine-tuning such filters, their effects become negligible as reported in the results section.

Summarily, assumed that there are sufficiently sophisticated seed processing techniques, aligners using fixed-size seeds and variable-size seeds should only differ with respect to runtimes.

METHODS

Informal Description

We first introduce our approach informally. Seeds can be visualized by drawing them as 45° lines in a plane that is spanned by reference (x-axis) and query (y-axis). Fig. 2 shows an example for this kind of seed visualization. A detailed description of this representation scheme of seeds can be found in [2]. If k-mers are visualized as proposed, some of them overlap or touch one another. Such overlapping or touching seeds can be merged into one single long seed. By performing all possible merges, we get the set of all MEMs. A major disadvantage of k-mers is the large size of the required hash-tables. For overcoming this problem, the concept of minimizers was introduced in [16]. Our proposed approach delivers MEMs for minimizers as well, if the endpoints of all merged k-mers are additionally extended. However, the length of this extension has a strict upper bound. Therefore, the proposed technique delivers long seeds of high entropy out of k-mers. The practical value of our technique is an observed runtime superiority regarding index creation and seed computation over the FMD-index as well as a reduction of the number of seeds compared to minimizers.

We now present the construction of MEMs out of minimizers. For this purpose, we first describe our technique in the context of k-mers with step size of one. Next, we extend the concept to m-step k-mers, before we finally show that it is applicable for minimizers as well.

Definitions and Notations

Let Σ be some alphabet with $\{\text{'A','C','G','T'}\} \subseteq \Sigma$ and let $W = w_0 w_1 \dots w_{n-1} \in \Sigma^+$ be a non-empty word over Σ. A substring $w_x w_{x+1} \dots w_{y-1}$ is denoted by $W[x,y)$. $\{(x, W[x, x+k)) : 0 \leq x \leq |W| - k\}$ is the set of all k-mer sequences over W and is denoted by W^k. Let $R, Q \in \Sigma^+$ be a reference and query sequence, respectively. A seed is a perfectly matching section between Q and R, which is represented as a triple (q, r, l), where q and r are the starting positions on Q and R, respectively $(q, r \in \mathbb{N}_0)$. l denotes the length of the seed $(l \in \mathbb{N})$. Let $s \coloneqq (q, r, l)$ be such a seed. Using $R\langle s \rangle$, we denote the subpart $R[r, r+l)$ of the reference that belongs to s. Accordingly, $Q\langle s \rangle$ denotes $Q[q, q+l)$. Hence, a seed describes the equivalence of $R\langle s \rangle$ and $Q\langle s \rangle$.

Given a reference R and query Q and their corresponding sets of k-mer sequences R^k and Q^k. Let $C^k \coloneqq \{Q^k \times R^k\}$ be the cross product of all query k-mer sequences and reference k-mer sequences over Q and R. Each pair $((q, W_Q), (r, W_R)) \in C^k$ with $W_Q = W_R$ defines a k-mer seed (q, r, k). The set of all such seeds over R^k and Q^k is denoted by $Seeds(Q^k, R^k)$. The δ-value of an arbitrarily sized seed (q, r, l) is defined as $r - q$. We call two seeds overlapping, if they have identical δ-values and overlapping or touching reference intervals. Trivially, this implies that overlapping seeds have overlapping query intervals as well.

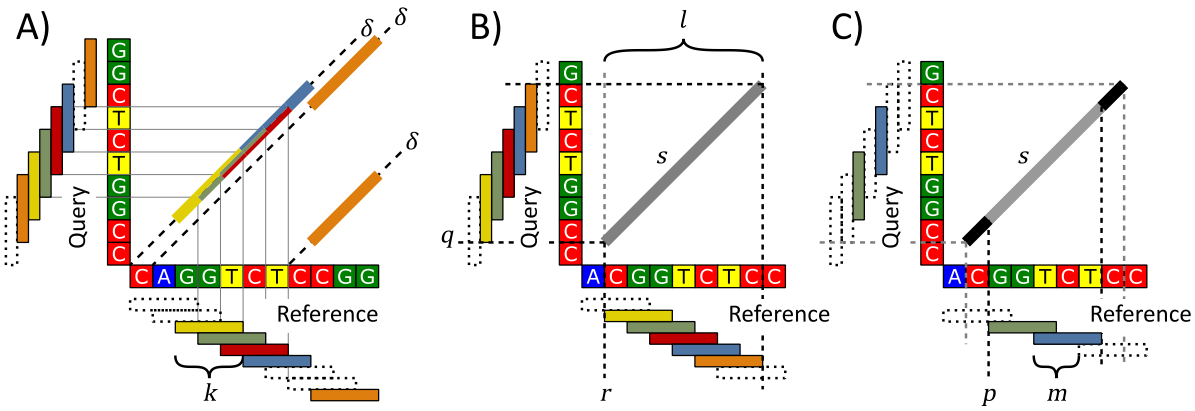

Figure 2. The three figures visualize several aspects of the MEM computation out of k-mers. **A)** In a plane spanned by reference (x-axis) and query (y-axis), overlapping k-mers form MEMs. The yellow, green, red and blue 3-mers form a single long MEM that spans 6nt. Overlapping k-mers are always on the same δ-line, which groups seeds according to their δ-value. **B, C)** In contrast to plain k-mers, m-step k-mers may merely form subparts of a MEM. The grey MEM $s = (q, r, l)$ is missing the first and last nucleotide in C), because the yellow and orange 3-mers are not part of the 2-step 3-mers. The black end sections of s require extension for being discovered.

Together, a set of overlapping k-mer seeds represents a larger region of equivalence between reference and query (see Fig. 2A). These larger regions correspond to the regions of equivalence described by the set of all MEMs. For a reference R and query Q, the set of all MEMs is defined, in accordance with [6], as follows: For all pairs of reference-query positions (x, y), with $R[x] = Q[y], 0 \leq x < |R|, 0 \leq y < |Q|$, we get a MEM (q, r, l) as follows: Let i be the minimal value for which $R[x - i - 1] \neq Q[y - i - 1]$. If there is no such value, then $i = \min(x, y)$. Using i, we set $q \coloneqq y - i$ and $r \coloneqq x - i$. Further, let j be the minimal value for which $R[x + j] \neq Q[y + j]$. If there is no such value, then $j = \min(|R| - x, |Q| - y)$. Using j, we set $l \coloneqq j + i$. The above definition can repeatedly create the same MEM. We assume that the set is purged of such duplicates.

Algorithmic Approach

Algorithmically, we can perform seed overlapping as follows: First, for a reference R and a query Q, we compute $Seeds(Q^k, R^k)$ and store these seeds in an array K. Then, we sort K according to the δ-values of all seeds. Seeds of equal delta value are sub-sorted according to their query position. By doing so, we guarantee that the sets of mutually overlapping k-mers appear consecutively in K. In an iteration over K, we combine overlapping k-mers. The time complexity of these operations is bounded by the sorting of K.

If the proposed algorithmic approach is applied to a set of k-mer seeds, it delivers a set of MEMs. Fig. 2A) visualizes the computation of three MEMs out of six 3-mer seeds. In the following, we extend our approach towards minimizers.

Extension of Algorithmic Approach to m-step k-mers

For doing so, we first introduce m-step k-mers. For a word W and a given step-size m, the set of all k-mer sequences fulfilling $x \bmod m = 0$ is called m-step k-mer sequences over W and is denoted by W_m^k.

We store the m-step k-mer seeds $Seeds(Q^k, R_m^k)$ in the array K. A comparison of Fig. 2C and Fig. 2B shows the effects of this modification: Two of the 2-step 3-mers cover parts of the seed s, but no k-mer reaches the endpoints of s anymore. Later we prove that each endpoint must be within a range of m nt to the closest k-mer. Via iterating diagonally and comparing query and reference for equality, we maximally extend the seeds in K. Alg. 1 implements the proposed approach using Pseudocode.

Algorithm 1. Computation of MEMs out of k-mers.

```
K-MERS_TO_MEMS(Q, R, K)
 1  SORT(K)                                   // sort by δ-value; for equal δ-value sort by q
 2  S = ∅; s = K[0]
 3  for j = 1 to K.length − 1                 // iterate over all k-mers
 4      k = K[j]                              // current k-mer
 5      if s.δ == k.δ and s.q + s.l >= k.q
 6          s.l = k.q + k.l − s.q             // enlarge s by k
 7      else S = S ∪ {s}; s = k               // start a new seed in S
 8  S = S ∪ {s}
 9  for all s ∈ S                             // maximally extend all seeds
10      while s.q + s.l < Q.length and s.r + s.l < R.length and Q[s.q + s.l] == R[s.r + s.l]
11          s.l = s.l + 1                         // extend rightwards
12      while s.q > 0 and s.r > 0 and Q[s.q − 1] == R[s.r − 1]
13          s.q = s.q − 1; s.r = s.r − 1; s.l = s.l + 1   // extend leftwards
14  return S                                  // S contains all MEMs
```

Please note, if we omit the seed merging (lines 2-8) and immediately extend all k-mers maximally (lines 9 -13), we obtain all MEMs but with two significant disadvantages: (1) The extension process is not limited in length anymore and its computational costs become proportional to the average size of MEMs. (2) A MEM containing n k-mers would be discovered n times. Therefore, we would need an additional purging scheme for getting rid of duplicates.

Formal Proof

We now formally prove that our approach successfully retrieves all MEMs out of m-step k-mers.

Lemma 1. Let Q be a query, R be a reference and $s \coloneqq (q, r, l)$ be a MEM over Q and R. If $l \geq k + m$, then there is at least one k-mer seed covered by s in $Seeds(Q^k, R_m^k)$.

Proof. Let $s' \coloneqq (q', r', k)$ be the sub-seed of s with $q' = q + x$, $r' = r + x$ for the offset $x = m - r \bmod m$ and the k-mer size k. By proving that s' is in $Seeds(Q^k, R_m^k)$, we prove the lemma: Let $(r', R\langle s'\rangle)$ and $(q', Q\langle s'\rangle)$ be two k-mers. Trivially, $(q', Q\langle s'\rangle) \in Q^k$ and $(r', R\langle s'\rangle) \in R^k$. We have to show that $(r', R\langle s'\rangle) \in R_m^k$, which immediately follows from $r' \bmod m = 0$. Finally, we have to show $Q\langle s'\rangle = R\langle s'\rangle$: Due to s, the words $R\langle s\rangle$ and $Q\langle s\rangle$ on reference and query must be equal. Considering $x \geq 0$ and $x + k \leq l$, the sub-words $R\langle s'\rangle = R[r + x, r + x + k)$ and $Q\langle s'\rangle = Q[q + x, q + x + k)$ must be equal as well. Altogether, s' results from $(q', Q\langle s'\rangle)$ and $(r', R\langle s'\rangle)$.■

Lemma 2. For a given MEM $s \coloneqq (q, r, l)$ over R and Q. For all pairs of seeds $s' \coloneqq (q', r', k)$, $s'' \coloneqq (q'', r'', k)$ in $K \coloneqq Seeds(Q\langle s\rangle^k, R\langle s\rangle_m^k)$, there is a chain of overlapping seeds in K connecting s' and s''.

Proof. By developing the claimed chain. Without loss of generality, we assume that $r' < r''$. Let $T \coloneqq \{(q + m * i, r + m * i, k) \in K \mid r' \leq r + m * i \leq r''\}$ be all seeds between s' and s". The seeds $(q + m * i, r + m * i, k)$ and $(q + m * (i + 1), r + m * (i + 1), k)$ are always overlapping (by $k - m$ nt) neighbors in T. Therefore, T represents the claimed chain. ■

Lemma 3. Let $s \coloneqq (q, r, l)$ be a MEM over R and Q. In $Seeds(Q^k, R_m^k)$, there exists at least one k-mer $s' \coloneqq (q', r', k)$ within m nt from r (i.e. $r + m \geq r'$) and at least one k-mer $s'' \coloneqq (q'', r'', k)$ within m nt of $r + l$ (i.e. $r'' + k + m \geq r + l$).

Proof. The existence of s' is proven by Lemma 1 already. The existence of the k-mer s'' can be proven by following the steps of Lemma 1 with the modification of choosing x as $l - k - (l + r - k) \bmod m$. ■

The following corollary follows directly from the Lemmas 1 and 2:

Corollary 1. For a reference R and a query Q and the k-mer seed set $Seeds(Q^k, R_m^k)$, Alg 1. computes all MEMs with a size $\geq m + k$ exactly once.

Extension to Minimizers and Extraction of SMEMs and Maximal Spanning Seeds

The proposed algorithmic approach for m-step k-mers can be applied to minimizers with a window of m, because the distance between two adjacent m, k-minimizers is always $\leq m$ [16]. Therefore, all argumentations regarding m-steps with k-mers still holds with minimizers.

Algorithm 2a. Computation of SMEMs out of MEMs.

```
MEMS_TO_SMEMS(M)
SORT(M)                                  // sort ascending by q; for equal q sort descending by l
S = ∅; qstart = 0; qend = 0
for i = 0 to M.length − 1                // iterate over all MEMs
    m = M[i]                             // m is current MEM
    e = m.q + m.l                        // end of m on query
    if e > qend                          // m is SMEM and query interval changed
        S = S ∪ {m}                      // add m to SMEMs
        qend = e; qstart = m.q           // update query interval
    elseif e == qend and m.q == qstart   // m is SMEM and query interval stays unchanged
        S = S ∪ {m}                      // add m to SMEMs
return S                                 // S contains all SMEMs now
```

Algorithm 2a shows the Pseudocode for extraction of SMEMs of MEMs using a sorting followed by a single sweep over all MEMs. In order to prove the correctness of the algorithm, we characterize SMEMs (in accordance to [6, 15]) as follows: A SMEM is a MEM that is not enclosed by another MEM on the query. The above algorithm identifies all enclosed MEMs by iterating over them ordered by their start positions on the query.

The variable $qend$ keeps the rightmost end-position of all MEMs visited during the iteration so far. For each iteration of the loop, we have: If the current seed m ends before $qend$, then m cannot be a SMEM, because the previously visited MEM that assigned $qend$ encloses m. $qstart$ memorizes the start position of the last SMEM. If start and end of the previous SMEM is equal to the current one (line 9), then the current seeds is a SMEM (line 10) that spans the same query interval as the previous one but a different reference interval. Alg. 2a requires time $O(n \log n)$ for a set of n MEMs, assumed the initial sorting is done in time $O(n \log n)$.

Algorithm 2b. Computation of Maximal Spanning Seeds out of MEMs.

```
MEMS-TO-MAXIMALLY-SPANNING(M)
 1  I = CONSTRUCT-INTERVAL-TREE(M)        // initialize interval tree using query intervals
 2  SORT(M)                               // ascending by query start
 3  S = ∅; qpos = 0
 4  repeat
 5      T = GET-OVERLAPPING(I, qpos)      // find all seeds with overlapping query interval
 6      if T.length > 0                   // are there overlapping seeds ?
 7          BUID-MAX-HEAP(T)              // descending by l; for equal l increasing by q
 8          s = EXTRACT-MAX(T)            // rightmost maximally spanning seed overlapping qpos
 9          S = S ∪ {s}
10          // check for further maximally spanning seeds
11          while T.length > 0 and s.l == MAXIMUM(T).l
12              S = S ∪ {EXTRACT-MAX(T)}// extract maximally spanning seed from heap
13          qpos = s.q + s.l              // update qpos to first position after s
14      elseif LOWER-BOUND-EXISTS(M, qpos)// are there still seeds beyond qpos ?
15          s = LOWER-BOUND(M, qpos)      // get the first seed beyond qpos
16          qpos = s.q                    // set qpos to end of uncovered area
17      // there are no more seeds beyond qpos. So, S contains all maximally spanning seeds now
18      else return S
```

Within the MEMs, we identify the set of maximally spanning seeds as follows: A MEM is a maximally spanning seed if and only if it comprises at least one query position, where it is not covered by another longer MEM. Please note that the maximally spanning seeds always represent a subset of the SMEMs. (A detailed comparison of SMEMs and maximally spanning seeds can be found in [2].)

We informally describe the approach of Alg. 2b, a detailed analysis is given in Supplementary Note 2. In the central loop (lines 4-18), we visit all maximally spanning seeds exactly once via $qpos$. In each iteration, we distinguish between two situations:

$qpos$ is within a set of seeds (Boolean condition in line 6 is true). We identify all maximally spanning seeds (line 8-12) by building a heap according to the seeds' lengths (line 7) for all seeds overlapping $qpos$ (line 5). If there are several maximally spanning seeds at $qpos$, then they are all identified in the while-loop (lines 11 and 12). Line 13 updates $qpos$ for the next iteration. The new value for $qpos$ guarantees that none of the maximally spanning seeds is discovered twice or lost.

$qpos$ is at the beginning of an area without seeds at all (Boolean condition in line 6 is false). In this case we move $qpos$ to the first seed after this area (lines 14-16) or we return the computed set S, because there are no unprocessed seeds anymore (line 18). If $qpos$ is moved to the start of a seed, we continue with case 1 in the next iteration.

Lemma 4. For a set of n MEMs, Alg. 2b) requires time $O(n \log n)$, assumed the sorting (line 2) and interval tree construction (line 1) happen in time $O(n \log n)$.

Proof. The central loop (lines 4-18) is executed $\leq 2n$ times. Further, lines 5, 14 and 15 require time $O(\log n)$ per iteration. In line 5, the accumulative size of all T is $\leq n$, because each MEM can only be part of a single iteration's T. Therefore, all operations performed on T (lines 7, 8, 11 and 12) must require an accumulative time of $O(n \log n)$. ■

RESULTS

We now analyze the proposed technique for computing MEMs, SMEMs and maximally spanning seeds out of minimizers considering practical circumstances. Using the human genome and the read generator Survivor [17] and DWGSIM [18], we generated reads of various sizes for subsections of the human genome. For benchmarking, we extended the MA code by integrating the Minimizer code of Minimap 2 as additional seeding module. Details regarding the benchmarking can be found in Supplementary Note 1.

We first analyze the effects of occurrence filters in the context of different seed generation schemes. Afterwards, we inspect the runtimes for various combinations of seeding approaches and index generation techniques. The benchmarking is followed by an analysis of the seed entropy for the different seeding techniques. Finally, we outline some use cases of our proposed approach.

Filter and Effects

Most aligners (e.g. Minimap 2 [3], MA [2], BWA-MEM [4]) use an filtering scheme in order to cope with repetitive regions of the genome. Using a threshold, seeds of query intervals that show an excessive amount of occurrences (i.e. a high ambiguity) on the reference are purged during seeding.

In order to accomplish comparability between FMD-index seeds and minimizers regarding this aspect, we rely on the flowing settings: For Minimap 2 we set '*-f*' to zero and the *'--min-occ-floor'* threshold to 200. For the FMD-Index, MA's *'Maximal Ambiguity'* parameter is set to 200. Therefore, both seeding schemes drop seeds occurring more the 200 times on the reference.

However, the FMD-Index filters once the extension is complete (using the size of the suffix-array intervals), while minimizers filter using the size of the hash-table buckets. This difference has an impact on the theoretical equivalence of the computed seed sets as shown in Fig. 3A. We now practically evaluated these differences in the context of the human genome.

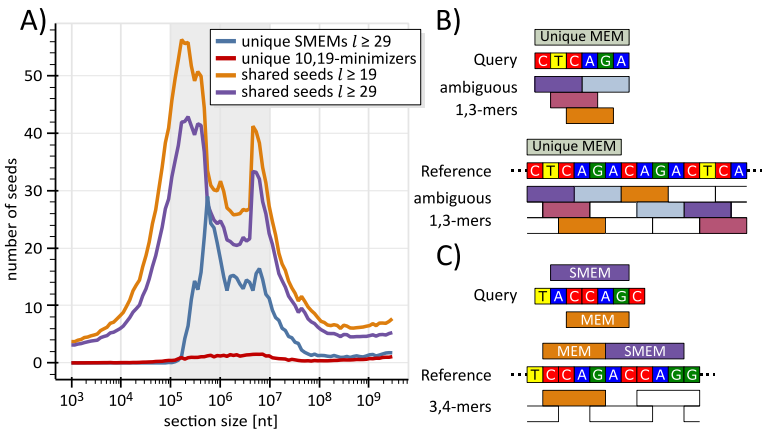

Figure 3. Subfigure **A)** reports about the number of shared and different seeds for 250nt Illumina reads for differently sized sections of the human genome, averaged over 10,000 reads. The red graph shows the absolute amount of seeds generated by Alg. 2a that do not exist in the set of SMEMs (with $l \geq 19$) computed by the FMD-index. The blue curve displays the opposite information; it shows all seeds (with $l \geq 29$) computed by the FMD-Index that do not appear in the seeds computed by Alg. 2a. The orange and purple graphs inform about the numbers of shared seeds for SMEMs and seeds computed by Alg. 2a for with $l \geq 19$ and $l \geq 29$, respectively. **B)** This subfigure gives an example of a MEM that is unique. However, all four $1,3$-mers within the MEM occur twice on the shown reference section. The location of all seeds on query and reference is indicated via colored boxes. **C)** This subfigure shows the generation of a "Pseudo-SMEM". The purple SMEM cannot be discovered using the $3,4$-mers, since no $3,4$-mer is contained within the seed. However, the orange MEM can be discovered using the orange $3,4$-mer. Now, Alg. 2a will not delete this MEM, since it is not enclosed by any other seed.

Without filtering, the red and blue graphs of Fig. 3A must always be zero. This equivalence is given in the region below 10^5nt, where the filter is still inactive. The movement of the red curve above zero shows the existence of seeds generated by Alg. 2a that are not SMEMs. The appearance of such "Pseudo-SMEMs" is visualized in Fig. 3C and can be explained as follows: Due to the absence of a minimizer, a SMEM cannot be found anymore. Instead, the largest MEM inside the query interval of such a SMEM appears as a Pseudo-SMEM now. The red curve shows that the amount of such seeds is within an acceptable range. The blue curve shows a more dramatic behavior and indicates a significant loss of seeds generated by Alg. 2a within the range of 10^5nt up to 10^8nt. This behavior can be explained by the difference with respect to filtering 19-mers and maximally extended seeds. In this area, we have many 19-mers that are already too ambiguous while the respective maximally extended seed is still below the occurrence threshold. Fig. 3B demonstrates this via an example. After 10^7nt even the maximally spanning seeds become too ambiguous, resulting in an overall decrease of seeds (orange and purple graphs) but also in a decrease of unique SMEMs. In Supplementary Note 4, we show the graphs of Fig. 3A for an increased occurrence limit of 1000. By increasing the limit, the sharp spike of the blue curve disappears almost completely for the price of a factor 4 regarding runtimes approximately.

Time Evaluation

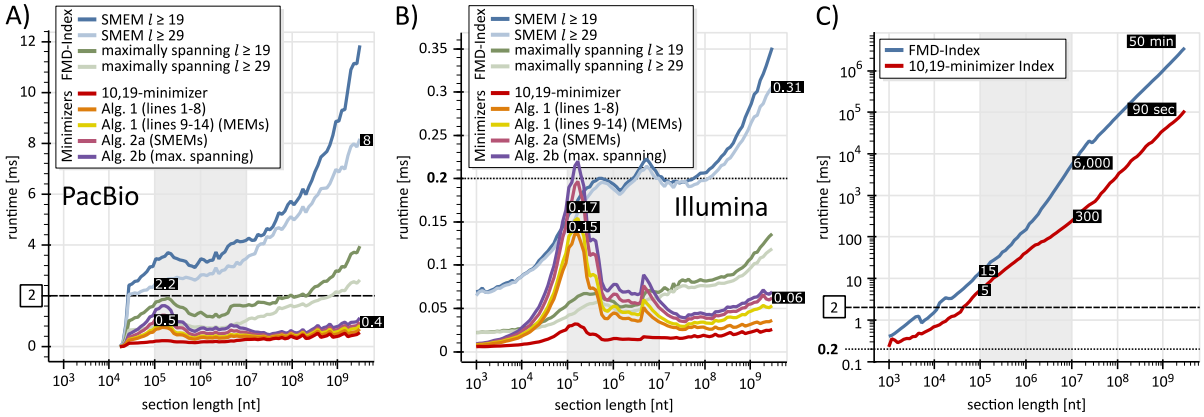

Figure 4. The graphs in A) show the measured runtimes for PacBio averaged over 1,000 reads. B) informs about the runtimes for Illumina averaged over 10,000 reads. The blue and green graphs reports about runtimes measured by using the FMD-index, while all other graphs show times for minimizers. All runtimes related to minimizers are accumulative. For example, the difference between the Alg. 2a and Alg. 1 (lines 9-14) informs about the additional time required for running Alg. 2a. Subfigure C) compares the observed runtimes for index creation between FMD-index and minimizers, where the scale of the y-axis is logarithmic. The x-axis is always of logarithmic scale and denotes the sizes of the subsections from the human genome used for runtime measurements. All details with respect to our measurements can be found in Supplementary Note 1.

Up to 10^5nt, the occurrence threshold stays inactive, because the picked sections of the human genome are too short to comprise sufficiently repetitive regions. Hence, this region shows the "unfiltered" behavior of minimizers and their worse asymptotic behavior compared to the FMD-index. As soon as the filter "kicks in", minimizers get significantly faster than the FMD-index. For PacBio as well as Illumina reads, it can be observed that the initial computation of minimizers and sorting (the orange graph) contributes most significantly to the overall runtime. The extension (Alg. 1, lines 9-14) contributes notably in the case of Illumina reads and large genome sizes only. The times required for maximum spanning seeds compared to SMEMs are significantly shorter with the FMD-index and slightly longer using minimizers (Alg. 2a and Alg. 2b). This can be explained as follows: With the FMD-index, the computation of maximally spanning seeds requires less extension efforts as well as reference position lookups in the suffix array than for SMEMs. In contrast, the filtering of MEMs requires more efforts for maximally spanning seeds than for SMEMs, because the SMEMs filter is less restrictive and less complex. Although the extraction of maximal spanning seeds requires an interval tree, a heap and an initial sorting operation, it is only slightly slower than the SMEM filtering. This can be explained by different behaviors of Alg. 2a and 2b in their central loops, where Alg. 2b merely iterates over maximal spanning seeds (and gaps among them) while Alg. 2a iterates over all MEMs. The Index generations for the FMD-index and minimizers show the same asymptotic behavior [16, 19]. Nevertheless, practically, the index generation for minimizers is magnitudes faster than for the FMD-index for reasonably sized genomes.

Seed Entropy Analysis

We now analyze the effects of the occurrence filtering with respect to the seed entropy for the different forms of seeding approaches. We define seed entropy as follows:

Definition: Let Q be a read originating from a reference interval I and let S be a set of seeds for Q. The seed entropy of S for Q is the ratio $n/|S|$, where n is the number of nucleotides in I that are covered by seeds of S and $|S|$ the size of the set S.

The entropy expresses the number of nucleotides that a single seed contributes to an accurate alignment on average. Expressed in other terms, the entropy is the ratio of correct data to incorrect data. Within the interval I, the proposed entropy does not distinguish between seeds contributing to an accurate alignment, we call them correct seeds, and all others in I. These incorrect seeds can erroneously increase the coverage. Therefore, our entropy represents an approximation merely. DWGSIM and Survivor, the read generators used here, do not deliver sufficient information for identifying correct seeds in I. Fig. 5C visualizes our entropy measurement scheme.

The graph for minimizers is always below the one for MEMs (computed by Alg. 1). This can be explained as follows: Combining seeds (Alg. 1, lines 1-9) reduces the number of seeds while maintaining the coverage. Additionally, extending on both ends (Alg. 1, lines 9-14) increases the coverage while keeping the number of seeds. Hence, Alg. 1 can only increase the seed entropy. Alg. 1 generates all MEMs with size ≥ 29nt and a subset of those that have a size between 19nt and 29nt. Therefore, all MEMs with $l < 29$ are deleted in the yellow curve, which naturally increases the seed entropy.

Generally, minimizers and MEMs deliver the worst seed entropy among all tested seed sets. Further, as already shown in [2], maximum spanning seeds deliver a higher seed entropy than SMEMs. In Fig. 5A and 5B, there is a difference between the curves for SMEMs computed using the FMD-index and SMEMs computed by Alg. 2a. The same difference can be observed with the curves for maximally spanning seeds. These differences originate from the occurrence filtering. If we decompose a repetitive region into many small fixed-size sections, the individual sections can all surpass an occurrence limit while the complete region does not. For such a region, all minimizers are purged but the MEM (computed by the FMD-index) is not. Aside from one correct position, such a MEM will occur on many other incorrect positions too (due to the repetitive region). This in turn hurts the overall seed entropy.

Our measurement expresses the entropy of a seed set and not its capability to allow accurate alignments. The proposed algorithmic approaches can only deliver seeds that are discoverable via minimizers and nothing beyond. Practically, however, the accuracy of minimap2 demonstrates that the minimizer filtering is not an issue.

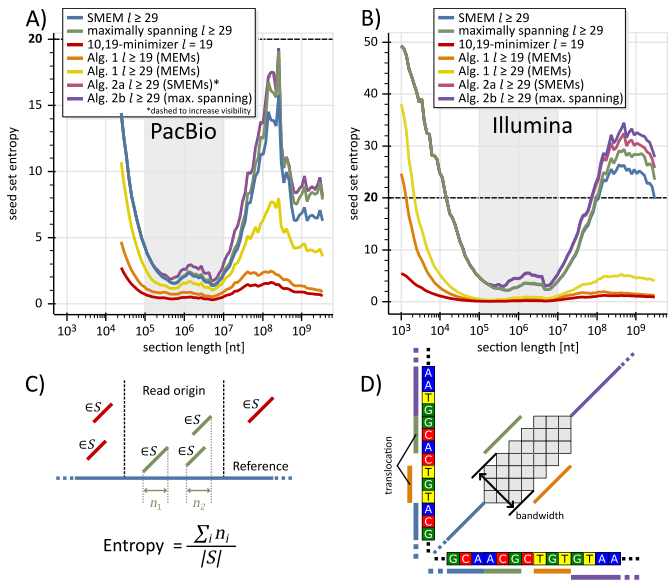

Figure 5. Subfigure **A)** analyzes the seed entropy for PacBio reads according to the entropy computation show in C). The data is averaged over 1,000 reads. The blue and green curves are for seeds that are generated using the FMD-index. The red curve is for the unprocessed minimizers, while all other curves represent benchmarks for the proposed algorithmic approaches. Likewise, **B)** shows the seed entropy analysis for Illumina reads averaged over 10,000 reads. **C)** Informally, for a read r, the entropy measures the ratio between the reference areas covered by "correctly placed" seeds of r over the total amount of seeds. **D)** visualizes a situation, where dynamic programming causes issues with recognizing the micro-translocation that could be discovered via reseeding in the affected area.

Applications of Approach

Fig. 4C shows that index computation for minimizers is significantly faster (e.g. 30 times for the entire human genome) than for the FMD-index. The short times for minimizers allow the application of "single-use" indices for seeding on subsections of the human genome. We now describe two scenarios for such a seeding:

For filling gaps between seeds, aligners often rely on Dynamic Programming [1-3, 12]. Technically, it is common to start from one seed's endpoint and to find a path to the next seed's start point throughout the matrix. There are structural variants that cannot be detected via such a path-oriented strategy. Examples of such SV are micro duplications and micro translocations as e.g. shown in Fig. 5D. There, the micro translocation can be discovered via the orange and green MEM above and below the band. Dynamic programming runs into trouble in such situations due to the out of band location of both MEMs as well as the limitation of the backtracking to incorporate only one of them into an alignments path. Due to their short length, a whole genome index is probably missing any minimizers indicating the orange and green MEM. Using a single-use index (with short minimizer sizes) for the gap, the two MEMs can be discovered quickly.

Due to the repetitiveness of genome sections, the initial seeding of a read might deliver a candidate region merely without precisely placing the read within such a region. In this case, a reseeding with smaller seeds can help acquiring a more accurate alignment. Like for the DP scenario mentioned above, indices for minimizers allow such a reseeding in short time and enable the fast delivery of MEMs, SMEMs and maximally spanning seeds via Alg. 2a, 2b.

DISCUSSION

Alg. 1 represents a surjective and not injective mapping from k-mers to MEMs (see Fig. 1). Therefore, informally spoken, everything that can be discovered with k-mers (fixed-size seeding) can also be discovered using MEMs (variable-size seeding) and vice versa. This implies that there is no theoretical superiority of one of these seeding techniques in contrast to assumptions made by us [2] and others [3]. A sophisticated chaining of minimizers should deliver the same alignment accuracy as a sophisticated chaining of MEMs. SMEMs and max. spanning seeds are subsets of MEMs. Therefore, these seed sets are incapable of delivering a higher accuracy than MEMs.

Let $|S|_C$ and $|S|_W$ be the absolute number of correct and incorrect seeds in a set of seeds S, respectively. We get the following hierarchies:

$$|k\text{-mers}|_C \geq |w,k\text{-minimizers}|_C \geq |\text{MEMs}|_C \geq |\text{SMEMs}|_C \geq |\text{max. spanning}|_C,$$
$$|k\text{-mers}|_W \geq |w,k\text{-minimizers}|_W \geq |\text{MEMs}|_W \geq |\text{SMEMs}|_W \geq |\text{max. spanning}|_W,$$

where MEMs, SMEMs and maximally spanning seeds are of size $\geq w + k$. Opposed to the above hierarchies, our benchmarking suggests a reverse hierarchy regarding the seed entropy, where the seed entropy is the expected number of nucleotides contributed to an accurate alignment by a single seed (see section 3.3). Hence, with max. spanning seeds, an aligner will use the most seed set with the highest entropy, but it is at high risk of missing an accurate alignment due to the lack of correct seeds. In contrast, with k-mers an aligner has access to the largest number of correct seeds but struggles to cope with the excessive amount of incorrect seeds.

The filtering of seeds according to their number of occurrences can disturb the above hierarchies: A correct MEM can be entirely composed of k-mers that surpass an occurrence limit, while the MEM itself does not. In this case, the MEM does not contribute to $|k\text{-mers}|_C$ but it contributes to $|\text{MEMs}|_C$. Our benchmarking indicates that this problem can be tackled by carefully adapting the filter limits for k-mers. However, a more sophisticated strategy is a “fall back” approach, where we switch to variable-size seeding in situations where fixed-size seeding fails.

Naturally, any approach that processes a set of seeds (e.g. chaining, clustering) has a complexity driven by the number of seeds to be processed. Therefore, processing minimizers is expected to be more expensive than processing MEMs, SMEMs or maximally spanning seeds, where the overhead depends on the characteristics of the processing scheme (e.g. non-heuristic chaining can have a squared worst case complexity [3]). In the context of this overhead, a preprocessing of seeds via our proposed algorithms can improve the overall runtime. A SoC computation [2] using minimizers is special here. In this case, Alg.1 does not require the initial sorting anymore (line 1, sort by δ-value), because the SoC can deliver the minimizers sorted by their δ-value already. Further, Alg. 1 can be directly integrated into the SoC computation. By doing so, the SoC can fulfill a double purpose by detecting candidate regions for alignments and simultaneously creating MEMs for these regions.

FMD-Index construction can be boosted by exploiting concurrency. However, trivially, such introduction of concurrency in the context of minimizers is possible as well. Here, we limit our benchmarking to a single thread view in order to get a fair comparison of both techniques. Considered the existence of the algorithms presented here, the excessive runtimes for index creation and seed extension indicate a limited value of the FMD-index in comparison to minimizers.

Our analysis bases on the human genome and a limited range of parameter combinations. A more comprehensive study could yield a better insight here. Further, the design of more sophisticated heuristics for filtering minimizers deserves additional research. Finally, the single-use index approach mentioned above lacks practical experience so far.

REFERENCES

1. Chaisson, M.J. and G. Tesler, *Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory.* BMC bioinformatics, 2012. **13**(1): p. 238.
2. Schmidt, M., K. Heese, and A. Kutzner, *Accurate high throughput alignment via line sweep-based seed processing.* Nature Communications, 2019. **10**(1): p. 1939.
3. Li, H., *Minimap2: pairwise alignment for nucleotide sequences.* Bioinformatics, 2018. **1**: p. 7.
4. Li, H., *Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM.* arXiv preprint arXiv:1303.3997, 2013.
5. Langmead, B. and S.L. Salzberg, *Fast gapped-read alignment with Bowtie 2.* Nature methods, 2012. **9**(4): p. 357-359.
6. Ahmed, N., K. Bertels, and Z. Al-Ars. *A comparison of seed-and-extend techniques in modern DNA read alignment algorithms*. in *Bioinformatics and Biomedicine (BIBM), 2016 IEEE International Conference on*. 2016. IEEE.
7. Brown, D.G., *A survey of seeding for sequence alignment*. Bioinformatics algorithms: techniques and applications. 2007. 126-152.
8. Abouelhoda, M.I. and E. Ohlebusch, *Chaining algorithms for multiple genome comparison.* Journal of Discrete Algorithms, 2005. **3**(2): p. 321-341.
9. Shibuya, T. and I. Kurochkin. *Match chaining algorithms for cDNA mapping*. in *International Workshop on Algorithms in Bioinformatics*. 2003. Springer.
10. Suzuki, H. and M. Kasahara, *Acceleration Of Nucleotide Semi-Global Alignment With Adaptive Banded Dynamic Programming.* bioRxiv, 2017: p. 130633.
11. Needleman, S.B. and C.D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal of molecular biology, 1970. **48**(3): p. 443-453.
12. Sović, I., et al., *Fast and sensitive mapping of nanopore sequencing reads with GraphMap.* Nature communications, 2016. **7**: p. 11307.
13. Ferragina, P. and G. Manzini. *Opportunistic data structures with applications*. in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. 2000. IEEE.
14. Liu, Y., T. Hankeln, and B. Schmidt, *Parallel and space-efficient construction of Burrows-Wheeler transform and suffix array for big genome data.* IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB), 2016. **13**(3): p. 592-598.
15. Li, H., *Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly.* Bioinformatics, 2012. **28**(14): p. 1838-1844.
16. Roberts, M., et al., *Reducing storage requirements for biological sequence comparison.* Bioinformatics, 2004. **20**(18): p. 3363-3369.
17. Jeffares, D.C., et al., *Transient structural variations have strong effects on quantitative traits and reproductive isolation in fission yeast.* Nature Communications, 2017. **8**(1): p. 14061.
18. Homer, N., *DWGSIM (2017).* URL https://github.com/nh13/DWGSIM, 2017.
19. Nong, G., S. Zhang, and W.H. Chan, *Two efficient algorithms for linear time suffix array construction.* IEEE Transactions on Computers, 2010. **60**(10): p. 1471-1484.