BMC Bioinformatics

RESEARCH

Open Access

PerFSeeB: designing long high-weight single spaced seeds for full sensitivity alignment with a given number of mismatches

Valeriy Titarenko[1*] and Sofya Titarenko[2]

*Correspondence: valeriy.titarenko@manchester.ac.uk

[1] School of Biological Sciences, University of Manchester, Oxford Road, Manchester M13 9PL, UK
[2] School of Mathematics, University of Leeds, Woodhouse, Leeds LS2 9JT, UK

Abstract

Background: Technical progress in computational hardware allows researchers to use new approaches for sequence alignment problems. For a given sequence, we usually use smaller subsequences (anchors) to find possible candidate positions within a reference sequence. We may create pairs ("position", "subsequence") for the reference sequence and keep all such records without compression, even on a budget computer. As sequences for new and reference genomes differ, the goal is to find anchors, so we tolerate differences and keep the number of candidate positions with the same anchors to a minimum. Spaced seeds (masks ignoring symbols at specific locations) are a way to approach the task. An ideal (full sensitivity) spaced seed should enable us to find all such positions subject to a given maximum number of mismatches permitted.

Results: Several algorithms to assist seed generation are presented. The first one finds all permitted spaced seeds iteratively. We observe specific patterns for the seeds of the highest weight. There are often periodic seeds with a simple relation between block size, length of the seed and read. The second algorithm produces blocks for periodic seeds for blocks of up to 50 symbols and up to nine mismatches. The third algorithm uses those lists to find spaced seeds for reads of an arbitrary length. Finally, we apply seeds to a real dataset and compare results for other popular seeds.

Conclusions: PerFSeeB approach helps to significantly reduce the number of reads' possible alignment positions for a known number of mismatches. Lists of long, high-weight spaced seeds are available in Additional file 1. The seeds are best in weight compared to seeds from other papers and can usually be applied to shorter reads. Codes for all algorithms and periodic blocks can be found at https://github.com/vtman/PerFSeeB.

Keywords: Spaced seeds, Lossless seed, Full sensitivity, Sequence alignment, Mismatch, Indexing

Background

Scientists use sequence analysis to understand organisms' features, structure, and function. By comparing sequences obtained for well-known and unexplored plants or animals, we may understand the biology of the investigated organisms. Currently, experimental techniques cannot provide us with whole sequences but only with many subsequences that we need to concatenate/merge in some way to form a long sequence. Thus, researchers have to solve a general sequence alignment problem before producing any meaningful conclusions related to the biological properties of an organism.

Suppose two long sequences are similar in some way. One sequence (a *reference* sequence) is known, and a researcher wants to find the other sequence. For example, as a reference sequence, we may use a human genome sequence (an "averaged" sequence based on genetic information of several individuals). We want to know a genome sequence for another person, i.e. a patient with a specific disease. The current hardware for genome sequence allows us to find only chunks of the unknown genome of the "patient". Those chunks (called *reads*) are usually relatively short (hundreds of base pairs) and may contain errors. We assume that the reference and "patient" sequences are similar, so we may use the reference sequence to align a set of reads accounting for possible mismatches.

The standard procedure is to consider each read and find its possible positions within the reference sequence such that the distance between the read and a part of the reference sequence is minimal. Subject to final goals, several definitions of distance as a measure of similarity between two sequences are possible. The distance may depend on all elements of sequences or only elements at specific locations, and it may also involve various transformations of sequences. The similarity between two strings was measured initially using dynamic programming algorithms, see [1–3]. However, due to time complexity, the use of these algorithms became impractical for the increased size of data available. While it is natural to position a read in a way to achieve the smallest distance, the found location may only sometimes be the best one.

With progress related to sequencing hardware and the amount of data provided, we may often simplify the original problem since many reads overlap. So for each position within a reference sequence, tens/hundreds of reads have regions similar to a chosen reference chunk. Ideally, we should align all reads so that there are fewer discrepancies between each other when aligned. Therefore for the initial step of aligning, for each read, we can measure the minimum distance (maximum similarity score) between the read and all possible positions within the reference sequence. Next, however, we should create a list of locations with slightly lower similarity scores. And for the next step, we need to account for all reads to achieve maximum similarity between reads sharing the same regions.

Choosing the best sequence alignment algorithm may also depend on an experimental technique used to acquire data. And data collection procedures tend to be prone to specific errors. For example, scientists usually fragment nucleic acid chains by physical, enzymatic and chemical approaches. However, the enzymatic process produces more insertions/deletions (*indels*) compared to physical techniques [4]. While everyone wants to acquire many high-quality long reads as quickly as possible and for a small cost, there is always a compromise: a large number of short (100–300 bp) reads with a reduced number of errors or a smaller number of long reads (> 1k bp) with extra errors [5]. Despite different possible approaches to defining distances between two sequences or similarity scores, we should expect that two close or similar sequences should have many identical elements ordered in the same way. Therefore we want to avoid checking all positions within a reference sequence, e.g. billions of positions for a human genome, but consider a limited number of candidate positions where a read and a reference sequence have common subsequences (anchors). We may be tempted to consider only those positions that provide us with the least distance from the read. However, more candidate positions with higher distance values may give us higher similarity scores when dynamic programming algorithms are applied. Therefore we need an algorithm to find possible candidate positions within a specific distance from a read.

In the late 90s, several new algorithms, e.g. BLAST [6, 7], appeared based on ideas of filtration and indexing. Researchers used short sequence fragments. They require pieces from the read to be present in a reference sequence to align a read. The search is sped up using various indexing, which may provide a researcher with "false-matching" positions. The fragments were considered contiguous segments. Once candidate positions based on the complete matching of contiguous regions are found, algorithms extend areas around the common parts to calculate a similarity score (seed-and-extend approach).

Originally only contiguous fragments were considered. However, in [8, 9], *spaced seeds* were introduced when elements of sequences were taken into account at specific positions only. We ignore possible different elements at other positions. For example, one popular spaced seed is 111010010100110111 introduced in [8]. Elements 1 mean that we take into account possible differences, while in the case of elements 0, we ignore them. The total number of 1 -elements is called the *weight* of a seed, and the total number of all elements of a seed is the *length*. For the above seed, the weight is 11 and length 18. Let us be given two sequences ATCATATCCGTAGCCTCT and ATCATAGCCGTTGCATCT of length 18. There are three mismatches (the seventh, twelfth and fifteenth elements from the left), so the sequences are different. However, when the above mask (seed) is applied, only eleven elements are compared, and all these elements are identical. Let the first string belong to a reference sequence and the second string belong to a read. Then the seed provides us with the same "signatures" for these strings. So if we have indexed all substrings in the reference sequence in the same way, we pre-align the read and perform an in-depth comparison after. However, if there are two strings ATCATATCCGTAGCCTCT and ACCGTATCGGTAACCTCT, then we have four mismatches (the second, fourth, ninth and thirteenth elements) and only of them are ignored. Therefore using the indexed library does not allow us to pre-align the read. The goal is to have such seeds that the corresponding indexed library is good for finding similar strings (allowing some number of mismatches) but, at the same time, does not have an excessive number of candidate locations to be checked later.

Some spaced seeds for different weights can be found in [10]. The idea of spaced seeds was extended for other problems: vector [11], indel [12], and neighbour [13] seeds. In ZOOM software [14], spaced seeds are generated to perform alignment with at most two mismatches. In PerM software [15], the authors used so-called *periodic spaced seeds* to improve mapping efficiency. Fast alignment-free string comparison based on spaced seeds (spaced-words) is discussed in [16]. Later approaches based on multiple seeds were used, e.g. [17, 18] (a good review can be found in [19]). Seeds in a multiple-seed environment are designed to have less overlap and thus increase the chances of hitting common regions. A good bibliography related to spaced seeds can be found in [20]. While papers for the original BLAST [6] and PSI-BLAST [7] algorithms have very high number of citations (more than 70 000 and 60 000), the developers of the modern versions of BLAST agree that the productivity of using spaced seeds is much higher compared to contiguous seeds [21]. SpeedBLAST is an example of an extension of original BLAST software based on spaced seeds which is superior in terms of efficiency, especially in the case of twilight zone hits [22]. Inspired by the results shown in [22] the authors consider the next step to be the modification of the BLAST algorithm to take advantage of the spaced seeds designed in this work. The full design of the algorithm for the local alignment is out of the scope of the current study.

Seed design may also deal with a non-binary alphabet. For example, in YASS [23] a three-letter alphabet (`#`, `@`, `-`) is used, where `#` stands for a nucleotide match, `@` is for a match or transition (`A` $\leftrightarrow$ `G` or `C` $\leftrightarrow$ `T` mutations) and `-` is used when we ignore corresponding symbols.

Computational resources available to researchers for the past 20 years have also improved significantly. For example, even for a budget computer it is possible to create a library of records, i.e. data structures of pairs (“key”, “value”). A human genome has a length of $\approx 3.2 \times 10^9 < 2^{32}$ bp. Therefore each position (“key”) within the sequence can be represented as a 32-bit number. Suppose the corresponding “value” for a given position depends on n elements around the position in a predefined order (n is the weight of a seed). If we avoid undefined areas, then each element is one of the following four symbols `A`, `C`, `G`, `T`, i.e. requires only 2 bits for storage, e.g. $\texttt{A} = \texttt{00}$, $\texttt{C} = \texttt{01}$, $\texttt{G} = \texttt{10}$ and $\texttt{T} = \texttt{11}$. Therefore each record (“key”, “value”) may require $(32 + 2n)$ bits. Firstly, we create a library of records and all these records are ordered by “key” elements which are unique for each record. Secondly, we sort all records by “values”. The library sorted in this way may have multiple records for the same index number (“value”). The total size of the library is at most $3.2 \cdot 10^9 \cdot (32 + 2n)$ bits or $8 \cdot 10^8 \cdot (16 + n)$ bytes. It is also possible to split the data into smaller chunks. For example, each chunk has the same first 16 bits of “values”, and we have $2^{16} = 65536$ chunks but shorter $(16 + 2n)$-bit records or $8 \cdot 10^8 \cdot (8 + n)$ bytes in total. So, for $n = 16$, the storage requirement is 17.9 GB, for $n = 32$, 48 and 64 we get 29.8, 41.7 and 53.6 GB, respectively.

The human genome contains repeated regions, so there may be extreme cases of many “keys” or no “keys” for specific “values”. However, by increasing the weight of a seed by one, we reduce the number of “keys” (positions) for each “value” four times. So, having high-weight seeds is our preference. On the other hand, the “patient” genome varies from the reference genome we know. Therefore the corresponding reads for the “patient” genome may also have mismatches, insertions, and deletions. Consequently, a candidate seed should cope with the presence of such deviations. In this paper, we only consider spaced seeds to deal with a known maximum number of mismatches (single-nucleotide polymorphisms or SNPs).

The sensitivity of seeds is usually measured using probabilistic approaches. In this paper, we consider an extreme case of *full sensitivity* seeds under the assumption that a read has at most n_m mismatches. This means that these seeds should allow us to find *all* candidate positions. Full sensitivity seeds are also known as *lossless seeds* [24] and were applied to filtration problems. A lossless filtration means that all fragments will be found (in the case of *lossy* filtration, we may miss some of them).

There may be several seeds of the maximum weight. Longer seeds require us to generate and check fewer index values in the library of records. We observed that most long high-weight seeds have a periodic structure. However, this structure is not a perfect one like discussed in [24] where a seed is made of several repetitions of structure A separated by structure B. We have an integer number of repetitions of structure A and a "remainder" of this structure (several first elements of structure A). The size of a periodic block has a simple relation with the lengths of reads and the seed. It is possible to generate those periodic blocks within a reasonable time frame (from a fraction of a second for blocks less than 30 symbols to several hours for cases of 50 symbols).

In the Methods section, we first discuss how to align reads to a reference sequence when an indexed library of records is used in the presence of possible mismatches. Then, we touch on ideas of software implementations affecting computational performance and how they dictate the choice of seeds. Methods to validate candidate seeds and approaches to speed up validation are proposed. Having a list of all valid short seeds, we attempt to design longer seeds and explain methods for seed extension. There may be too many valid seeds, and it is hard to construct them all using the seed extension approach. Therefore for long reads, it is better to create so-called periodic seeds made of several same concatenated blocks and a "remainder". Rules to form these blocks, as well as whole seeds to be valid, are discussed. In the Results section, we summarise the properties of periodic seeds, list the most popular spaced seeds and compare them with seeds designed with the PerFSeeB approach. Software tools are available to create library of records for given seeds and find candidate positions for reads. By applying the PerFSeeB approach to a real dataset, we show that one should check significantly fewer candidate positions for several possible mismatches.

Methods

Sequence alignment

Let there be an alphabet $\mathcal{A}$ of symbols and two strings x and y of length n consisting of characters from $\mathcal{A}$. The term distance $d(x, y)$ between two strings x and y was introduced in [25] for binary alphabet $\mathcal{A} = \{0, 1\}$ in the space of 2^n points as the number of mismatches between points x and y. In general, we may introduce a list of elementary operations which convert a source string x into a target string y. Each function may have a different cost. The distance is then a sum of all costs. If we cannot transform x into y, then the cost is ∞. A good review of possible distances can be found in [26]. The most common distances are the following ones.

1. Levenshtein (or edit) distance (insertions, deletions, substitutions are allowed at equal cost of 1) [27]. For example, we may transform word "health" into "shale" as "health" → "healt" → "heale" → "sheale" → "shale" (2 deletions, 1 substitution, 1 insertion), so $d(\text{"health"}, \text{"shale"}) = 4$.
2. Hamming distance (only substitutions) [28].
3. Longest common subsequence distance (insertions/deletions) [29].

Suppose we have two sequences $x = \texttt{CTTGTCGTTGGAGATCGGAAGAGCA}$ and $y = \texttt{TAGGTGCTCG}$ of length $n_1 = 25$ and $n_2 = 10$, respectively. We define $m = n_1 - n_2 + 1 = 16$ and for each index $j = 1, \ldots, m$ we find distance $d_j = \sum_{k=1}^{n_2} \delta(x_{j+k}, y_k)$, where $\delta(a, b) = 1$ if symbols a and b are different, otherwise $\delta(a, b) = 0$. For example, to find d_8 we align the corresponding strings and calculate values for $\delta(x_{j+k}, y_k)$:

```
string 1  CTTGTCGTTGGAGATCGGAAGAGCA
string 2  ________TAGGTGCTCG_______
δk        ________0100101000_______
```

So, $d_8 = 3$, in a similar way we get $d = (6, 5, 9, 7, 6, 9, 9, 3, 7, 8, 7, 8, 8, 8, 7, 7)$. We compare all n_2 symbols for both sequences. However, by introducing spaced seeds we may ignore some symbols. For example, we may use seed of length n_2: `1011011111` (binary notation used in [8]) or `#-##-#####` (notation used in [9]), where `1` or `#` stand for a match and `0` or `-` for 'do not care'. Thus, to find the distance d_8 accounting for the spaced seed we get

```
string 1  CTTGTCGTTGGAGATCGGAAGAGCA
string 2  ________TAGGTGCTCG_______
δk        ________0100101000_______
seed      ________#-##-#####_______
new δk    ________0000001000_______
```

or $d_8 = 1$. Similarly, we get $d = (5, 3, 7, 6, 5, 7, 7, 1, 5, 6, 7, 6, 7, 6, 5, 5)$. From now, we will use binary notation as it is closer to binary numbers utilized for storage and computation.

Now let us consider the library of records. Suppose we have a contiguous seed $s = \texttt{1111}$ of weight $w = 4$. So, if we take the first four symbols of vector x, then we encode string `CTTG` as $1 + 4 \cdot 3 + 4^2 \cdot 3 + 4^3 \cdot 2 = 1 + 12 + 48 + 128 = 189$ (if we set $\texttt{A} = 0, \texttt{C} = 1, \texttt{G} = 2, \texttt{T} = 3$). Therefore the library of records for the reference sequence x and seed s has 22 pairs ("key", "value"):

$(1, \texttt{CTTG} = 189)$	$(2, \texttt{TTGT} = 239)$	$(3, \texttt{TGTC} = 123)$	$(4, \texttt{GTCG} = 158)$
$(5, \texttt{TCGT} = 231)$	$(6, \texttt{CGTT} = 249)$	$(7, \texttt{GTTG} = 190)$	$(8, \texttt{TTGG} = 175)$
$(9, \texttt{TGGA} = 43)$	$(10, \texttt{GGAG} = 138)$	$(11, \texttt{GAGA} = 34)$	$(12, \texttt{AGAT} = 200)$
$(13, \texttt{GATC} = 114)$	$(14, \texttt{ATCG} = 156)$	$(15, \texttt{TCGG} = 167)$	$(16, \texttt{CGGA} = 41)$
$(17, \texttt{GGAA} = 10)$	$(18, \texttt{GAAG} = 130)$	$(19, \texttt{AAGA} = 32)$	$(20, \texttt{AGAG} = 136)$
$(21, \texttt{GAGC} = 98)$	$(22, \texttt{AGCA} = 24)$		

Similarly, we may find "values" for sequence y:

$$\texttt{TAGG} = 163, \texttt{AGGT} = 232, \texttt{GGTG} = 186, \texttt{GTGC} = 110,$$
$$\texttt{TGCT} = 219, \texttt{GCTC} = 118, \texttt{CTCG} = 157.$$

We may see that no "values" for y can be found in the library generated for x. Therefore the use of "seed and extend" approach does not allow us to align y with respect to x, since no candidate positions can be found.

Suppose we use seed $s = \texttt{101011}$ (also of weight 4). We generate a new library for x (now containing 20 pairs):

$(1, \texttt{CTTC} = 125)$	$(2, \texttt{TGCG} = 155)$	$(3, \texttt{TTGT} = 239)$	$(4, \texttt{GCTT} = 246)$
$(5, \texttt{TGTG} = 187)$	$(6, \texttt{CTGG} = 173)$	$(7, \texttt{GTGA} = 46)$	$(8, \texttt{TGAG} = 139)$
$(9, \texttt{TGGA} = 43)$	$(10, \texttt{GAAT} = 194)$	$(11, \texttt{GGTC} = 122)$	$(12, \texttt{AACG} = 144)$
$(13, \texttt{GTGG} = 174)$	$(14, \texttt{ACGA} = 36)$	$(15, \texttt{TGAA} = 11)$	$(16, \texttt{CGAG} = 137)$
$(17, \texttt{GAGA} = 34)$	$(18, \texttt{GAAG} = 130)$	$(19, \texttt{AGGC} = 104)$	$(20, \texttt{AACA} = 16)$

For sequence y we get "values":

$$\texttt{TGTG} = 187, \texttt{AGGC} = 104, \texttt{GTCT} = 222, \texttt{GGTC} = 122, \texttt{TCCG} = 151.$$

There are three "values" (187, 104 and 122) found in the new library. Therefore we need to try three positions: $5 - 1 + 1 = 5, 19 - 2 + 1 = 18, 11 - 4 + 1 = 8$. For the first position (5) we get

```
string 1  CTTGTCGTTGGAGATCGGAAGAGCA
string 2  ____TAGGTGCTCG___________
δk        ____0101001111___________
```

and $d_5 = 6$, the second position (18) cannot be used, since the aligned string y will be out of the range for string x and the third position (8) gives us

```
string 1  CTTGTCGTTGGAGATCGGAAGAGCA
string 2  _______TAGGTGCTCG________
δk        _______0100101000________
```

and $d_8 = 3$. So, the seed `101011` allows us to find the best alignment position of y with respect to x.

We are interested in aligning genetic sequences. Reference sequences may consist of several separate sequences. For example, T2T_CHM13v2.0 Telomere-to-Telomere assembly (see [30]) contains 24 separate sequences of four symbols `A`, `C`, `G`, `T`. However, GRCh38.p14 release has 705 separate sequences (chromosomes, genomic scaffolds and patches) and five symbols (`A`, `C`, `G`, `T`, and `N` for unknown symbols). We may always concatenate separate sequences into one long sequence by adding extra "void" symbols in between to avoid forming library records containing symbols from different original sequences. In the case of the T2T reference genome, we may use two bits to encode each symbol. For the GRCh38.p14 reference genome, we have five possible values to combine every three consecutive symbols into a 7-bit number (as $5^3 = 125 < 128 = 2^7$). So, if we do not use any data compression, then storing the T2T sequence may require 0.73 GB, and for GRCh38.p14, we need about 0.84 GB. These numbers are minimal, even for a budget computer. Therefore we may use more storage space to achieve better performance. The 2-bit encoding is preferable even if we need to exclude some substrings containing any character except `A`, `C`, `G`, `T`.

Modern computers allow users to exploit SIMD (single instruction, multiple data) properties of CPUs (central processing units). For example, one instruction can be applied to a 128-, 256- or 512-bit number. CPUs can do various logical and shift operations very fast. A list of Intel's SIMD intrinsics designed for various CPU instruction sets can be found in [31]. Most computers (servers, workstations and home computers) built for the past decade support the 128-bit instruction set. Thus, we will use 128-bit instructions. There are similar instructions for 256- and 512-bit numbers, so the ideas discussed in the paper can also be applied for new architectures.

We may split the reference sequence into groups of 32 symbols, then form 128 bits such that the first 32 bits are 1s if the corresponding symbols are `A`, the second 32 bits are for symbols `C`, the third and fourth groups of 32 bits are for `G` and `T`, respectively. For example, the string `CATAGNCACGTGATCCTAGNCATGTTACCTGT` of 32 symbols has the following components

```
 sequence CATAGNCACGTGATCCTAGNCATGTTACCTGT 32-bit number
    A     01010001000010000100010000100000 0x0422108a
    C     10000010100000110000100000011000 0x1810c141
    G     00001000010100000010000100000010 0x40840a10
    T     00100000001001001000001011000101 0xa3412404
 A|C|G|T  11111011111111111110111111111111 0xfff7ffdf
```

If an element is `N` symbol, then the corresponding bits of all four 32-bit numbers are zeros. We use symbol "`|`" for logical OR operation, similarly, symbol "`&`" is for logical AND operation.

Let there be two sequences of length 32. We want to find how many symbols are the same for them (if one symbol is `N`, then there is no match).

```
    m1    CATAGNCACGTGATCCTAGNCATGTTACCTGT
    m2    GCCTCAGTTTTCACTCTATCAATATGTAATAA
 m1 & m2  ___________T_A__CTA___AT_T____T__
```

There are nine such symbols. We may apply logical AND operation for corresponding 32-bit components of `m1` and `m2`

```
              A          C          G          T
    m1    0x0422108a 0x1810c141 0x40840a10 0xa3412404
    m2    0xd8b21020 0x0008a816 0x02000041 0x25454788
 m1 & m2  0x00221000 0x00008000 0x00000000 0x21410400
```

Since `0x00221000 | 0x00008000 | 0x00000000 | 0x21410400 = 0x21639400`, then the number of 1-bits in `0x21639400` equals 9 and is the total number of matches. Now we approach the main goal of the manuscript, i.e. how to design seeds that will allow us to find candidate positions.

Choice of seeds

Suppose there is a short sequence we want to align (*read*) for a known long *reference* sequence. The length of the read is n_r. We may always assume that a read contains only four symbols (`A`, `C`, `G` and `T`). While there may be cases of reads containing symbols `N` for unknown letters, the number of such cases is often negligible. Let there be a seed s of length n_s (total number of all bits, 1s and 0s) and weight w (the number of 1s). If we have 1-bit, then the corresponding symbol for a given sequence is taken into account; otherwise (in the case of 0-bit), it is ignored. We may also assume that a seed's first and last bits are 1-bits. *Contiguous seeds* have only 1-bits, and the lengths and weights of these seeds are equal. *Spaced seeds* have at least one 0-bit.

Seeds allow us to form a shorter sequence from a longer one. We apply a seed of length n_s to a sequence of the same length and form a new sequence of length w when symbols of the original sequence are in front of 1-bits of the seed. We consider the new shorter sequence as a characteristic/signature of the longer sequence. In the case of a 4-letter alphabet, we may perform one-to-one mapping of the short sequence and a $2w$-bit number. We use these numbers to find pairs of ("key", "value") in the library of records generated for the reference sequence. These pairs should have "value"-components equal to our $2w$-bit numbers. We may generate only $(n_r - n_s + 1)$ $2w$-bit numbers for each read since the first and last bits of a seed should be within the read's boundaries.

Suppose we have a seed of weight four, and by applying it at some reads' position, we get "value" `ACGT`. Let there be N pairs in the reference library with the same "value". Now we expand the original seed and assume the new seed at the chosen position will also be within reads' boundaries. This means that the new seed may provide us with four new "values" (`ACGTA`, `ACGTC`, `ACGTG`, `ACGTT`). The number of ("key", "value") pairs found for the new seeds will be less or equal to N. For example, if the reference sequence is `AGTGACGT`, then we have one pair for `ACGT` and no pairs for the 5-symbol sequences. Of course, it might happen that the number of pairs in the library generated for "value" `ACGTA` is N, and there are no pairs found for `ACGTC`, `ACGTG` and `ACGTT`. However, we may often think that the number of pairs for each 5-symbol sequence is around $N/4$.

For a library of records generated for a reference sequence we may estimate the total number of pairs found for a given "value" as $\alpha/4^w$, where α is a constant. We generate $(n_r - n_s + 1)$ numbers ("values") for a read. Based on the above empirical rule, we should expect to consider about $\alpha(n_r - n_s + 1)/4^w$ candidate positions in the reference sequence where we try to align the given read. Each "value" is found for a substring of a read with a given shift. Since the goal is to pre-align the read, the "keys" we found should be corrected by the corresponding shifts of a substring to the read's first element. For example, if a read has an exact match within a read, then all its "values" should also have an exact match. So, in the best scenario, we should have about $\alpha/4^w$ unique candidate positions (when a read can be placed exactly at several locations of the reference sequence). However, it is better to estimate the number of candidate positions from the above as

$$\alpha \frac{n_r - n_s + 1}{4^w}. \tag{1}$$

This number indicates what steps we should perform to reduce the number of candidates' locations and thus improve performance of sequence alignment algorithms. The steps are

1. find seeds of maximum weight,
2. among the found seeds choose seeds of maximum length.

Of course, the trivial approach is to maximise n_s and w. So, a contiguous seed of length n_r is the best candidate. However, there are restrictions as we should account for various mismatches between the reference and "patient" sequences. In this paper we consider only pointwise mismatches (SNPs) and suppose all reads have at most n_m mismatches. Hereafter, we assume that all seeds we try to find meet this rule.

- Let there be two arbitrary reference and read sequences such that the read can be aligned with no more than n_m mismatches. Then at least one "value" for the read should be paired with the records from the library generated for the reference sequence.

Suppose there are two seeds s_1 and s_2 and we may align s_1 with respect to s_2 in such a way that all 1-bits of seed s_1 are also present in s_2. Then s_1 is a *subset* of s_2 and can be denoted as $s_1 \subset s_2$. For example, if $s_1 = \texttt{10111}$, $s_2 = \texttt{11011101}$, then by shifting s_1 by one element to the right we get

```
s2 11011101
s1 _10111___
```

Note that there may be multiple possible shifts and s_1 may have more 0-bits. For example, $s_3 = \texttt{10101} \subset s_1$ and we get two possible alignments

```
s2 11011101
s3 _10101___
s3 ____10101
```

Seed $s_4 = \texttt{1101011} \not\subset s_2$.

The human genome has many repeated regions, so several candidate seeds have similar weights/lengths. Therefore, it is reasonable to choose a seed with a more uniform distribution of 1-bits rather than seeds with grouped 1-bits. In any case, if we have many repeated experiments providing us with reads of the same length for a known reference sequence, it is worth performing a statistical analysis of the "keys" distribution to avoid cases when for some "values" we have thousands of "keys".

According to [24], we may consider two main alignment problems: *lossless* alignment when we detect *all* locations in the reference sequence and *lossy* alignment when we may miss some of them. As for any statistical analysis, we may have true-positive (TP), true-negative (TN) and false-positive (FP) and false-negative (FN) events. We aim to find seeds that provide us with lossless alignment. Therefore for a given read and seed, we construct all "values" according to the procedure described above, and then we do not miss any candidate location. Thus the number of false negative events is zero. Since

$$\text{sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \tag{2}$$

then for our seeds FN = 0 and sensitivity = 100%. So, our seeds have *full sensitivity* or we have *lossless* seeds.

Seed validation

Let us consider an example. We check if seed $s = \texttt{110101011001011}$ is a full sensitivity seed for reads of length 20 and at most two mismatches. The length of s is 15. Therefore there are $(20 - 15 + 1) = 6$ possible positions of the seed with respect to a read. We shift the seed and pad it with zeros (five extra zeros for each row). Thus we get the following six rows of length 20:

$$\begin{aligned}
L_1 &= \mathtt{11010101100101100000},\\
L_2 &= \mathtt{01101010110010110000},\\
L_3 &= \mathtt{00110101011001011000},\\
L_4 &= \mathtt{00011010101100101100},\\
L_5 &= \mathtt{00001101010110010110},\\
L_6 &= \mathtt{00000110101011001011}.
\end{aligned}$$

A seed is *valid* if for any two positions of mismatches within a read, there is at least one row of the matrix with both 0-elements at the given columns. For example, we choose columns 7 and 12, then we may pick up the third row with both 0-elements (underlined):

$$\mathtt{001101\underline{0}1011\underline{0}01011000}$$

Note that 0-elements can be from the original seed or the padded ones. For example, if we choose columns 5 and 16, then the last row of the matrix has both 0-elements (the first 0-element is the padded element):

$$\mathtt{0000\underline{0}1101010110\underline{0}1011}$$

However, if we choose columns 4 and 13, then for each row of the matrix there is at least one 1-element:

$$\begin{array}{l}
\mathtt{110\underline{1}01011001\underline{0}1100000}\\
\mathtt{011\underline{0}10101100\underline{1}0110000}\\
\mathtt{001\underline{1}01010110\underline{0}1011000}\\
\mathtt{000\underline{1}10101011\underline{0}0101100}\\
\mathtt{000\underline{0}11010101\underline{1}0010110}\\
\mathtt{000\underline{0}01101010\underline{1}1001011}
\end{array}$$

Therefore seed $s = \mathtt{110101011001011}$ is not a valid full sensitivity seed for $n_m = 2$, $n_r = 20$. To check validity we should check all possible combinations, for the above example we have to check $C_{20}^2 \equiv 20!/2!(20-2)! = 190$ cases, where $m! \equiv 1 \cdot 2 \cdot 3 \cdots m$. For example, we have two sequences AAAAAAAAAAAAAAAAAAAA (reference sequence) and AAATAAAAAAAAATAAAAAAA (read). For the reference sequence we create its library of records:

$$\begin{array}{lll}
(1, \mathtt{AAAAAAAAA}), & (2, \mathtt{AAAAAAAAA}), & (3, \mathtt{AAAAAAAAA}),\\
(4, \mathtt{AAAAAAAAA}), & (5, \mathtt{AAAAAAAAA}), & (6, \mathtt{AAAAAAAAA}),
\end{array}$$

i.e. all "values" are the same. The read provides us with the following six "values":

$$\begin{array}{lll}
\mathtt{AATAAAAAA}, & \mathtt{AAAAAATAA}, & \mathtt{ATAAAAAAA},\\
\mathtt{TAAAAAAAA}, & \mathtt{AAAAATAAA}, & \mathtt{AAAATAAAA}.
\end{array}$$

There is no read's "value" equal to "values" in the library.

Now consider a general case. Let there be a seed s of length n_s. We want to check if the seed meets the full sensitivity requirement for any read of length n_r and a maximum number of n_m mismatches. Suppose n_m and n_r are set. Then we create n_m-vector of indices i_k, $k = 1, 2, \ldots, n_m$ such that $1 \le i_k \le n_r$. By definition, the length of a seed is not greater than the length of a read, $n_s \le n_r$. We may shift the first element of a seed by δ with respect to the first element of the read. As the last element of the seed should be within the read's elements, then δ can vary from 0 to $(n_r - n_s)$. For each value of δ, we check that none of the 1-elements of the seed shifted by δ has indices i_k, $k = 1, 2, \ldots, n_m$. If for any possible combination of indices i_k, there is at least one value of δ (depending on i_k, $k = 1, \ldots, n_m$), the requirement is met, then the seed is a *valid* seed for given n_m and n_r.

Parameter δ has $(n_r - n_s + 1)$ values. By padding the seed vector with 0-elements from left and right, we can form $(n_r - n_s + 1)$ vectors of length n_r. We pad from left and right and the total number of 0-elements to be used for each vector is $(n_r - n_s)$. One may perform padding differently; however, it may be more convenient to have the first element of the seed aligned with the first element of the read for the first vector. The next vectors are just the previous vectors padded by one 0-element from the left (as is done above). Or we may align the last elements of the seed and read and pad by 0-element from the right.

In any case, for a given seed, we form $(n_r - n_s + 1)$ vectors of length n_r. We generate all possible combinations of i_k indices. There are n_m indices, and we want them to be different. Thus we get

$$C_{n_r}^{n_m} \equiv \frac{n_r!}{n_m!(n_r - n_m)!} \tag{3}$$

such combinations. If, for any of these combinations, all $(n_r - n_m + 1)$ vectors/rows contain at least one 1-element for chosen indices i_k, then the seed cannot be used. Otherwise, the seed meets the requirements, i.e. it is a valid seed.

Now we discuss how to perform seed validation as vector operations. As it is shown above, by padding seed s of length n_s with 0-elements from the left and right, we create $(n_r - n_s + 1)$ rows L_p of length n_r. If there are n_m mismatches, then we need to process $C_{n_r}^{n_m}$ cases. We create a vector V for each case, so all its elements are 1s except n_m elements, which are 0-elements. For the first example, we choose elements 7 and 12, so one can write the corresponding V_1 vector as $V_1 = \texttt{11111101111011111111}$. Then one needs to check that $V|L_p$ equals V for at least one index $p = 1, \ldots, (n_r - n_s + 1)$. For the above example, we check L_3:

$$\begin{aligned}
V_1 &= \texttt{11111101111011111111},\\
L_3 &= \texttt{00110101011001011000},\\
V_1|L_3 &= \texttt{11111101111011111111},
\end{aligned}$$

so we get $V_1|L_3 = V_1$. Note that logical OR operations are bitwise, i.e., applied to each element of a vector.

The third example (elements 4 and 13) gives us vector $V_3 = \texttt{11101111111101111111}$, then for all vectors L_p, $p = 1, \ldots, 6$, we find $V_3|L_p$:

$$\begin{aligned}
V_3|L_1 &= \texttt{11111111111101111111} \neq V_3,\\
V_3|L_2 &= \texttt{11101111111111111111} \neq V_3,\\
V_3|L_3 &= \texttt{11111111111101111111} \neq V_3,\\
V_3|L_4 &= \texttt{11111111111101111111} \neq V_3,\\
V_3|L_5 &= \texttt{11111111111111111111} \neq V_3,\\
V_3|L_6 &= \texttt{11111111111111111111} \neq V_3.
\end{aligned}$$

Therefore, one cannot use the seed for the given combination of indices, and, as a result, it cannot provide us with full sensitivity. Only when the procedure is successful for all $C_{n_r}^{n_m}$ combinations, then the seed is valid.

It is clear that if we pad all vectors L_p and V with 1-elements, then the validation criterion is still valid since $1|1 = 1$. Therefore we may use various SIMD intrinsics to perform logical OR operations on 128-bit numbers.

There is an alternative approach to validating seeds. We create n_r binary vectors (columns) U_t, $t = 1, \dots, n_r$, of length $(n_r - n_m + 1)$, see Fig. 1. As before, we may pad them with 1-elements to form numbers of a given length, e.g. 32-, 64- or 128-bit numbers. The task is to consider all $C_{n_r}^{n_m}$ combinations of columns U_t, perform logical OR operations, i.e.

$$U_{j_1}|U_{j_2}|\dots|U_{j_{n_m}}, \tag{4}$$

and check if the resultant column has all 1-elements (let us call it the *saturated* vector).

Now we try to reduce the number of columns. We may identify the same columns and leave only different ones (remove 10 out of 36 columns in the example in Fig. 2). So, as $n_m = 4$ and $n_r = 36$, then instead of $C_{36}^4 = 58905$ combinations, we should check only $C_{26}^4 = 14950$. The next step is identifying those columns that are subsets of other columns. We denote $U_k \subset U_m$ if $U_k|U_m$ is U_m. This means that all 1-elements of column U_k are also 1-elements of U_m (however, there may be positions such that element q of U_m is 1-element but element q of U_k is 0-element). For example, $U_{14} = (\mathtt{0010000100})^T$, $U_{19} = (\mathtt{0010001100})^T$ and $U_{14} \subset U_{19}$. If there is a combination of vectors U_k that includes U_{14} and provides us with the saturated resultant vector, then the same combination but with U_{19} also provides us with the saturated vector. So, we may exclude U_{14}. Removing columns that are subsets of other columns allows us to decrease the number of combinations further. For the example in Fig. 2, we excluded extra 16 columns, so the total number of combinations is $C_{36-10-16}^4 = C_{10}^4 = 210$, i.e. the number of combinations we need to check is 280 times fewer compared to the original case.

We may use similar approaches when forming resultant columns. Suppose $n_m > 2$. As $U_{i_1}|U_{i_2} = U_{i_2}|U_{i_1}$ and $U_{i_1}|U_{i_1} = U_{i_1}$, then we vary i_1 from 1 to $(n_r - 1)$ and then i_2 from $(i_1 + 1)$ to n_r, so we consider $n(n-1)/2$ combinations. U_t are binary vectors, so the

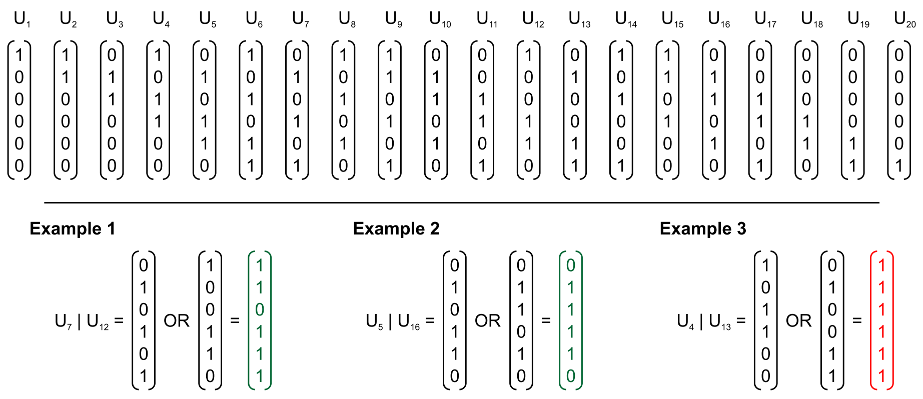

U1 U2 U3 U4 U5 U6 U7 U8 U9 U10 U11 U12 U13 U14 U15 U16 U17 U18 U19 U20
Example 1
U7 | U12 =
OR
Example 2
U5 | U16 =
Example 3
U4 | U13 =

Fig. 1 Seed validation based on columns. When the requirement is met, the row is in green colour, otherwise, in red colour

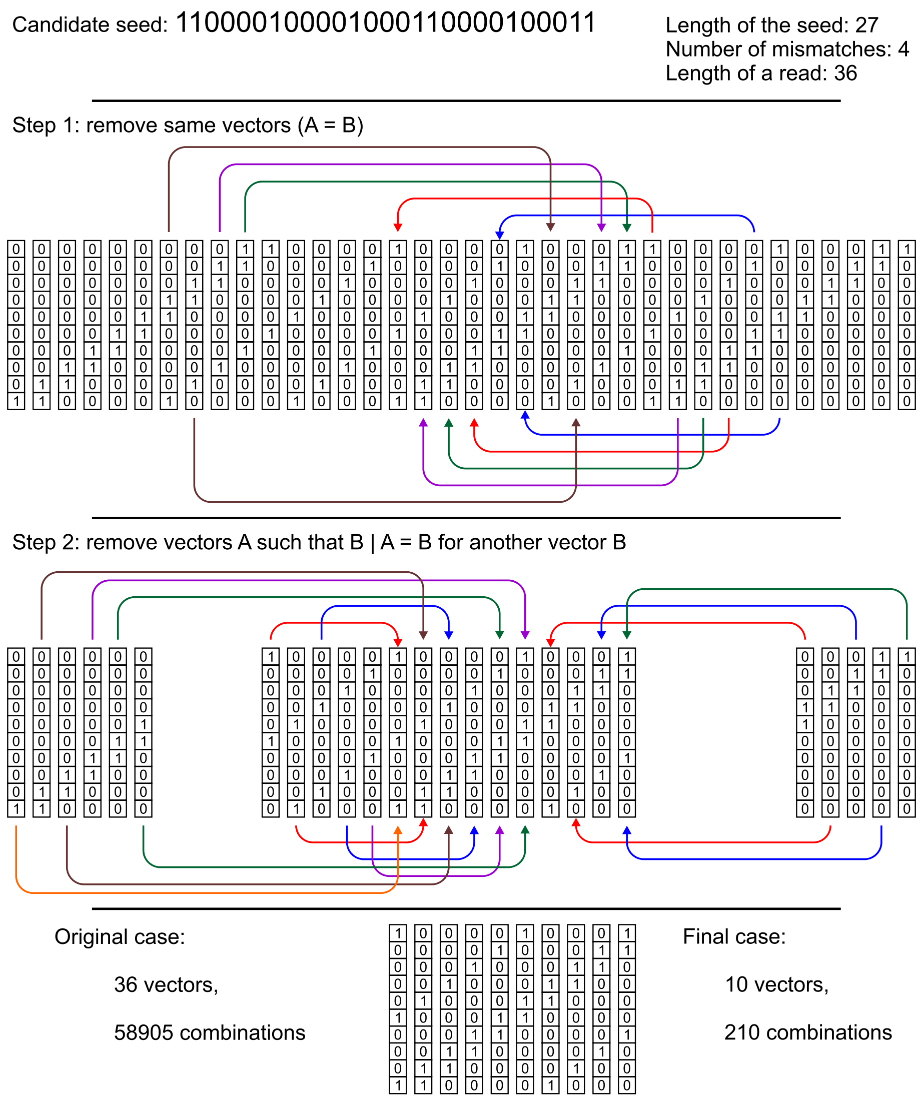


Fig. 2 Reducing the number of combinations when checking seed's validity

resultant vectors or OR operations. So all vectors can be considered as numbers in binary representation and sorted in ascending/descending order. Likely, some of the $n(n-1)/2$ resultant vectors are the same, so we exclude them. Therefore by keeping resultant vectors for intermediate steps and checking if new vectors (i.e. U_{i_3}) are subsets of vectors from the set of intermediate resultant vectors (i.e. $U_{i_3} \subset \left(U_{i_1} | U_{i_2}\right)$), we may speed up processing.

Seed expansion

The first and last elements of a seed are 1-elements. There is only one seed of length 1 (`1`), one seed of length 2 (`11`), two seeds of length 3 (`101`, `111`), four seeds of length 4 (`1001`, `1011`, `1101`, 1111). Therefore for a read of length n_r, there are

$$1 + 1 + 2 + 4 + \ldots + 2^{n_r-2} = 2^{n_r-1} \tag{5}$$

seeds in total. As validation of each seed is also quite time-consuming, we should identify ways to reduce the number of candidate seeds. For example, seed `1000111010010001 1101` is valid for $n_r = 30$ and $n_m = 3$. Any subset of this seed is also valid (seeds of a lower weight). So, `1110100100011101` (the first four elements of the original seed are removed), `100011101001000111` (the last two elements), `10001010100100010101` (two random 1-elements are removed) are also valid seeds. Therefore, we can implement the following procedure.

1. Suppose we have already found all valid seeds of length less or equal to k.
2. We pad all these seeds with 0-elements from their right ends, so we get vectors of length k.
3. We concatenate them with 1-element (also at the right ends of the padded seeds).
4. Before applying the validation procedure, we form a seed by removing the first 1-element and all 0-elements from the left end and checking that the seed is in the list of valid seeds.

For example, if an original valid seed was `100111011` (of length 9), and we aim to find valid seeds of length 13, then the extended seed is `1001110110001`, and we need to check if `1110110001` is already in the list of valid seeds.

Clear that a seed is valid if and only if the reverse seed (the order of elements is reversed) is valid. So, seeds `1001110110001` and `1000110111001` are valid (or are not valid) simultaneously.

Periodic seeds

The above procedure allows us to reduce the number of candidate seeds. However, there are still a lot of seeds to be validated, so finding all possible seeds for a read's length of more than 45 is very slow. However, we can observe a very important property. When for given n_r and n_m we find all seeds of maximum weight, then there are almost always *periodic* seeds such that

$$n_r = n_s + T - 1, \tag{6}$$

where T is the size of the periodic block. Such seeds have a whole number n_b of these periodic blocks and the "remainder" (the first $n_d > 0$ elements of the block), so $n_s = n_b \cdot T + n_d$.

For example, for $n_r = 17$ and $n_m = 3$ we get three pairs of seed: `111011`, `1101000011`, `1100100011` (and the reversed seeds). For the first seed, we have `111011` $=$ `1110` $+$ `11`, so $T = 4$ and $n_s = 6$. We check that $n_s + T - 1 = 6 + 4 - 1 = 9 \neq 17 = n_r$. However, for the second seed we get `1101000011` $=$ `11010000` $+$ `11`, $T = 8$, $n_s = 10$, so $n_s + T - 1 = 10 + 8 - 1 = 17 = n_r$. Thus, for four seeds Eq. (6) is met but for other two seeds it is not.

There may be seeds of different period T for the same length n_r of a read. For example, in Fig. 3 there are seeds found for $n_r = 43$, $n_m = 4$. The maximum weight is $w = 12$. We get two pairs of seeds for $T = 19$, three pairs for $T = 17$ and one for $T = 13$.

Note that when one can find such seeds, other (shorter) seeds are often available. There are also several exceptions from the observation. For example, for some values

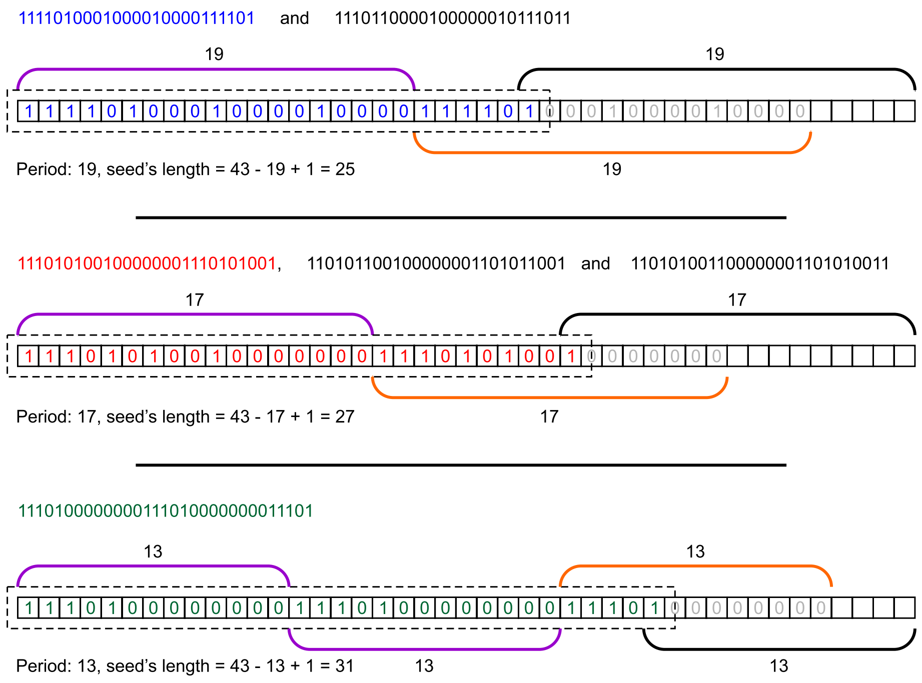

1111010001000010000111101 and 1110111000010000001011101 1
19
19
Period: 19, seed's length = 43 - 19 + 1 = 25
19
111010100100000000111010100 1, 110101110010000000110101100 1 and 110101001100000000110101001 1
17
17
Period: 17, seed's length = 43 - 17 + 1 = 27
17
1110100000000111010000000011101
13
13
Period: 13, seed's length = 43 - 13 + 1 = 31
13
13

Fig. 3 All seeds of maximum weight (12) found for reads of length 43, number of mismatches is 4. Reversed seeds are not shown

Table 1 Exceptions from the observed formula

n_m	n_r, **min**	n_r, **max**	**Exceptions (**n_r **values)**
2	12	42	no
3	11	42	11 (×), 13 (12), 39 (38)
4	20	52	no
5	13	50	13 (12), 16 (×), 17 (×), 19 (18), 20 (×), 47 (46)
6	14	53	15 (14), 17 (16), 19 (18), 23–26 (×), 34 (×), 38 (×), 39 (×), 47 (×), 48 (×), 52 (×)
7	16	46	17 (16), 19 (18), 21–23 (×), 25 (24), 26 (×), 27 (×), 29 (28), 43 (×), 45 (44)
8	18	46	19 (18), 21 (20), 23–26 (×), 28–32 (×), 34 (33), 42–44 (×)

If the formula is valid for shorter reads, then the corresponding n_r is in parentheses, otherwise (×)

of n_r and n_m, we may have seeds such that (6) is valid for smaller values of n_r, e.g. $n_r - 1$, $n_r - 2$. However, there may be cases when the formula is not true for the best seeds. See Table 1. For example, if $n_r = 16$, $n_m = 5$ the best seeds are `1101` and `1011` ($n_s = 4$ and $T = 3$), so $n_s + T - 1 = 4 + 3 - 1 = 6 \neq 16$. However, for 80% of read lengths, formula (6) is valid for a given value of n_r, for 87% is true for smaller values of n_r.

Suppose there is a periodic seed such that $n_s = n_b T + n_d$ and formula (6) is true. We want to check if a seed is valid for a given n_m; see Fig. 4. We need to generate indices j_k, $k = 1, \dots, n_m$ such that $1 \leq j_k \leq n_r$ and check if the resultant column $U_{j_1} | U_{j_2} | \dots | U_{n_m}$ is the saturated vector. A periodic block should meet the same requirements if the seed is valid. For this purpose, we choose j_k such that $(1 + T) \leq j_k \leq 2T$. Now we

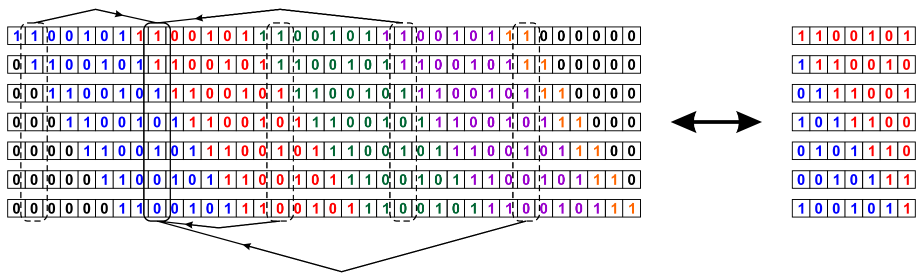


Fig. 4 A seed is valid only when its periodic block is valid

assume that the resultant column for the periodic block is not the saturated one for all possible combinations of indices. For any index $j_k, 1 \leq j_k \leq n_r$, we may write down $j_k = j_k^* + m \cdot T$, where $(1 + T) \leq j_k^* \leq 2T$ and m is an integer number. Therefore $U_{j_k} \subset U_{j_k^*}$ and instead of U_{j_k} we may consider $U_{j_k^*}$.

Periodic blocks

We validate a periodic block as the whole seed:

- generating all possible combinations of indices j_k, $k = 1, \ldots, n_m$, such that $1 \leq j_k \leq T$;
- checking if the resultant column is the saturated one.

A periodic block is valid if and only if its reverse is valid. We may also perform a cyclic rotation of elements of the block; those new blocks are valid at the same time. As a result, we have groups of $2T$ periodic blocks of length T (some may be identical) that are valid/not valid simultaneously. We call them *equivalent* blocks. Therefore it is enough to consider the validity of only one block. To reduce the number of blocks to be considered, we require that the first element is always 0-element and the last element is always 1-element. See examples of equivalent blocks generated for a given periodic block in Fig. 5.

We need a procedure to choose only one block instead of $2T$ blocks. Of course, for some blocks (e.g. `00101101`), there may be cases of identical blocks obtained via reversion/cyclic rotation. By varying indices j_k, we create patterns of 0-elements. Those patterns should be within the periodic block (or one of the equivalent blocks obtained by cyclic rotation) since there should be a row such that the corresponding elements of the chosen columns are all 0-elements. Therefore each periodic block must have a contiguous chunk of n_m 0-elements. Thus when we generate various periodic blocks, we may always assume that the first n_m elements are 0-elements.

To choose one block, we pick up the one with the highest number of 0-elements at the beginning. Among blocks obtained for the reverse block, there will also be a block with the maximum number of 0-elements. Then, we need to perform a further comparison. We may count the number of 1-elements in contiguous blocks from the left/right side of the zero-block and choose the block with fewer 1-elements. If the contiguous blocks

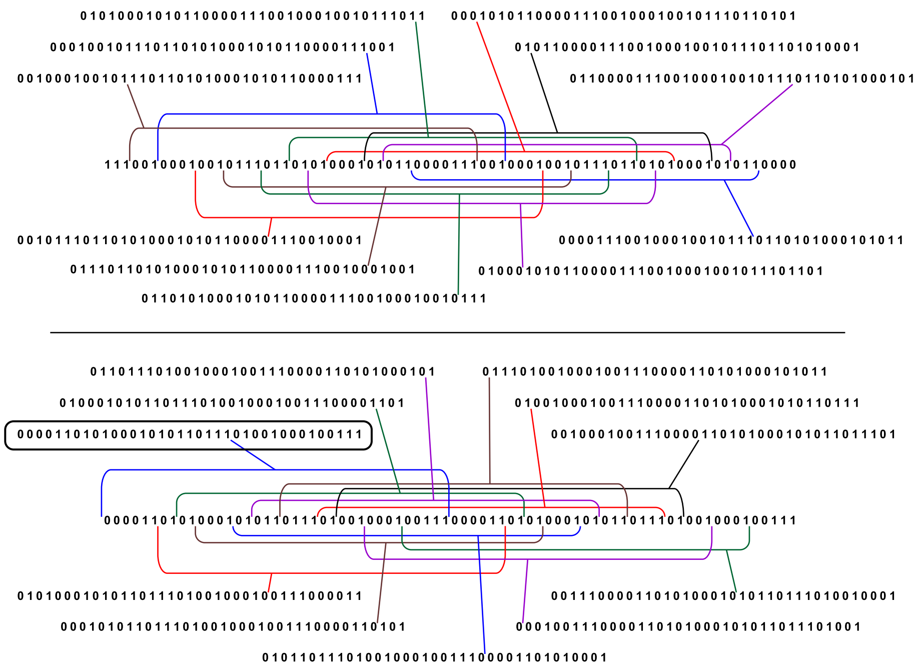


Fig. 5 Equivalent periodic blocks generated for block 101110100100010011100001101010001010 1. Only blocks with the first 0-element and last 1-element are shown. The bottom part is for the reverse block. The circled block is the one we choose from the group

are the same, then consider neighbouring blocks of 0-elements (and choose the smallest one). We repeat the procedure if it is needed.

Algorithm 1 allows us to validate a block, i.e. check if it can be used for reads of at most n_m mismatches. Suppose we have a binary vector s of length T. By performing a cyclic rotation of the vector when the last n elements of the vector become the first elements of a new vector (the order of elements is preserved), we generate T vectors η_i of length T. Then we just need to consider all possible combinations of n_m vectors and check if the resultant vector obtained by applying logical bitwise OR operation is the saturate vector, i.e. the vector with all 1-elements. For this purpose we create n_m indices k_p, $p = 1, \ldots, n_m$. When we have n_m binary vectors and apply OR operation, then the order of vectors does not matter. Therefore we may assume that $k_p < k_{p+1}$, $p = 1, \ldots, (n_m - 1)$. So, we initialise the indices as $k_p = p$, $p = 1, \ldots, n_m$, and to create a new set of indices we increment the last index k_{n_m}. When k_{n_m} reaches the maximum value (T), we start incrementing the previous index k_{n_m-1} and reset k_{n_m} to the smallest value permitted. This procedure is applied in a similar way to other indices. Clearly, that if after incrementing of k_p by one we get k_p equal to $(T + p - n_m + 1)$, then we should increment k_{p-1} are reset all other indices $k_p, k_{p+1}, k_{p+2}, \ldots$.

Algorithm 1 Checking is a binary vector s of length T is valid for n_m mismatches

```
procedure VALIDATEBLOCK($s$, $n_m$)
    $\mu_0 \leftarrow 0$, $\mu_1 \leftarrow 1$                    ▷ $\mu_0$ and $\mu_1$ are binary vectors of length $T$
    $L \leftarrow T - n_m + 1$, $u_1 \leftarrow \mu_0$
    for $i \leftarrow 1, T$ do                                    ▷ cyclic shift
        for $j \leftarrow 0, T-1$ do
            $\eta_i[j] \leftarrow s[(i+j)\%T]$             ▷ % is modulo operator, remainder
        end for
    end for
    for $i = 1, n_m$ do
        $k_i \leftarrow i$, $u_{i+1} \leftarrow u_i|\eta_{k_i}$             ▷ Bitwise logical OR operation
        if $u_{i+1} = \mu_1$ then return false
    end for
    $p \leftarrow n_m$, $K \leftarrow k_p$
    loop
        loop
            $K \leftarrow K + 1$
            if $K - p \neq L$ then break
            $p \leftarrow p - 1$
            if $p = 1$ then return true
            $K \leftarrow k_p$
        end loop
        $u_{p+1} \leftarrow u_p|\eta_K$, $k_p \leftarrow K$
        if $u_{p+1} = u_p$ then continue
        while $p < n_m$ do
            $p \leftarrow p + 1$, $K \leftarrow K + 1$, $i_p \leftarrow K$, $u_{p+1} \leftarrow u_p|\eta_K$
            if $u_{p+1} = u_p$ then break
        end while
        if $u_{p+1} = \mu_1$ then return false
    end loop
end procedure
```

The algorithm should stop when $p - 1 = 0$. However, we may stop earlier when $p - 1 = 1$. This means that we may actually always set $k_1 = 1$. Suppose that we have an arbitrary combination of ordered indices k_p, $p = 1, \ldots, n_m$ and form a resultant vector v after applying bitwise OR operations. However we may create another set of indices $\bar{k}_p$ such that $\bar{k}_p \equiv (k_p - k_1)\%T + 1$. The resultant vector found for k_p indices is the resultant vector found for $\bar{k}_p$ indices but after the cyclic shift by $(k_1 - 1)$ elements was performed. So, both resultant vectors are (or are not) saturated at the same time.

We store intermediate n_m resultant vectors u_p and perform bitwise OR operation with the corresponding η_{k_p} to find u_{p+1}. If η_{k_p} is a subset vector for u_p, then as it was discussed in previous subsections we may ignore η_{k_p} and consider other vectors.

As we want to find seeds of maximum weight, it is reasonable to find periodic blocks of maximum weight. Suppose there are blocks found for weight w. By replacing 1-elements with 0-elements we form periodic blocks of weight $(w - 1)$. However, there may also be blocks of weight $(w - 1)$ that cannot be formed by replacing 1-elements in blocks of weight w. For example, there are five blocks for $T = 11$, $n_m = 2$ and weight $w = 7$: `00011011111`, `00101011111`, `00101110111`, `00101111011`, `00110101111`. Block `00010110111` of weight $w = 6$ cannot be formed from those blocks. Seeds are formed of an integer number of periodic blocks and a "remainder". Therefore, there is a possibility that the best seeds are formed of blocks of non-maximum weight. However, when we generated seeds of non-maximum weight, they never formed seeds of weight larger than seeds formed from maximum-weight blocks.

Thus to generate all seeds of maximum weight for blocks of length T and n_m mismatches we set maximum weight w (at most $T - n_m$) and consider all binary vectors such that the first n_m elements are zeros and the last element is one. Then validate these seeds using Algorithm 1. If no seeds are found for a given w, then reduce w by 1 and generate a new set of candidate vectors.

All ideas mentioned above allow us to reduce the number of blocks to be validated. It is also possible to parallelise processing so each CPU thread validates only specific seeds from a pre-generated list. Together with SIMD instructions for validation steps, we sped up the generation of periodic blocks. It may take less than a second for $n_r < 35$, however, finding periodic blocks for $n_r \approx 50$ may still take hours as trillions of blocks should be validated.

Algorithm 1 for validation of seed blocks can be implemented on various computational architectures. The authors implemented it using various SIMD operations. The algorithm's performance will depend on input parameters and possible optimisation done by a compiler and specific elementary operations. This algorithm has five major operations: OR, XOR, binary shift and extraction of a number applied to 128-bit numbers (all SIMD operations) and elementary addition (as increment/decrement operations) for 32-bit numbers. We may ignore the time spent on other operations. In principle, the number of XOR, binary shift and extraction operations is the same for this algorithm. So, validating a single block will take about $O(n_{\mathrm{OR}} + 3n_{\mathrm{XOR}} + n_{add})$ elementary operations. Thus, we mention only OR (n_{OR}), XOR (n_{XOR}) and elementary addition (n_{add}) operations. These numbers are in Table 2. We may see that the ratio $n_{\mathrm{XOR}}/n_{\mathrm{OR}}$ is almost the same (≈ 0.92), while n_{add}/n_{OR} is increased with n_m (from 0.3 for $n_m = 2$ to 0.9 for $n_m = 9$). The total number of OR operations may differ for different T; however, we usually see a five-fold increase when n_m is increased by one. The main issue when generating all possible seed blocks is exponential (as a function of T, different functions for different values of n_m, e.g. $\sim T^{12}$ or T^{20}) increase of the number of blocks to be validated even after various procedures to reduce the number of candidate blocks.

The times needed to generate periodic blocks depend on CPU architecture. Run times for Intel Core i5-9600K processor (6 cores, 12 threads, base frequency 3.70 GHz) can be found in Table 3 and Fig. 6. One may see that calculations for $n_m = 4$ are the slowest, and in this case, the run times increase exponentially as $2.85 \cdot 10^{-10} \cdot 2.03^T$ (in seconds). If blocks are less than 30, we can perform all calculations in less than a second. When blocks have 40 elements, we need an hour to complete the task, while 50 elements may require us to spend a week or use a CPU with tens of cores.

Forming periodic spaced seeds

The final step of PerFSeeB approach (periodic full sensitivity blocks) is to form spaced seeds of maximum weight. For this purpose, we consider all periodic blocks

Table 2 Number of operations used for Algorithm 1: number of mismatches (n_m), length of a periodic block (T), total number of seeds to be validated (# tests), averaged numbers of SIMD OR (n_{OR}), XOR (n_{XOR}) and standard addition (n_{add}) operations for each binary block as an absolute number or as percentage of n_{OR} operations

n_m	**T**	**# tests**	n_{OR}	n_{XOR}	**% of** n_{OR}	n_{add}	**% of** n_{OR}
2	30	2391	218	194	89	74	34
2	35	6259	174	150	86	51	29
2	40	66,542	206	182	88	67	33
2	50	330,586	161	137	85	45	28
2	60	7,780,954	169	145	86	49	29
3	25	6296	459	419	91	188	41
3	29	88,578	955	906	95	464	49
3	33	595,263	387	350	90	145	37
3	38	14,780,813	571	532	93	241	42
4	25	4989	2858	2724	95	1630	57
4	29	68,931	3405	3274	96	1893	56
4	33	966,747	4292	4153	97	2361	55
4	38	32,240,220	1501	1437	96	730	49
5	25	780	105,377	97,501	93	77,081	73
5	29	13,668	30,507	28,945	95	19,946	65
5	33	229,001	14,869	14,342	96	8865	60
5	38	9,574,775	6432	6238	97	3566	55
6	25	283	80,726	74,048	92	61,606	76
6	30	7520	32,220	30,638	95	21,071	65
6	35	200,437	29,377	28,254	96	18,128	62
6	38	502,937	320,811	305,715	95	208,510	65
7	25	83	176,631	158,692	90	146,921	83
7	30	6205	7026	6672	95	4362	62
7	33	5173	717,404	665,819	93	526,252	73
7	38	448,976	31,566	30,395	96	19,488	62
8	30	728	92,676	85,827	93	70,045	76
8	35	1909	32,199,261	29,015,989	90	26,230,401	81
9	30	177	2,632,939	2,267,998	86	2,547,863	97
9	35	1784	1,722,183	1,556,851	90	1,419,823	82

formed for a given number n_m of mismatches. As $n_s = n_b T + n_d$, $n_b \geq 1, 1 \leq n_d < T$, and $n_r = n_s + T - 1$, then $n_r = n_b T + n_d + T - 1 = (n_b + 1)T + (n_d - 1) \geq 2T$ or $T \leq n_r/2$. For each periodic block we form all equivalent blocks and check if

1. the first symbol of the equivalent block is 1-element,
2. the n_d-th element is also 1-element.

In principle, there is no need to consider blocks obtained from the reverse periodic block as the final seed will be a reverse of seeds formed from the original block. Once the seed is formed we count its weight. By processing all periodic blocks we find seeds

Table 3 Calculation times (in seconds) for Algorithm 1 for Intel Core i5-9600K processor (n_m is the number of mismatches, T is the size of the periodic block)

T	$n_m = 2$	$n_m = 3$	$n_m = 4$	$n_m = 5$	$n_m = 6$	$n_m = 7$	$n_m = 8$	$n_m = 9$
25	0.02	0.02	0.02	0.03	0.02	0.02	0.03	0.01
30	0.05	0.69	1.10	0.28	0.16	0.05	0.04	0.35
35	0.05	10.45	16.11	5.87	2.43	2.09	15.82	1.08
38	0.41	73.85	141.16	48.25	45.46	5.73	22.41	3.64
40	0.63	48.71	551.05	203.14	106.70	27.52	32.51	8.40
42	0.93	849.18	2257.21	809.38	235.92	88.09	53.71	34.19
45	1.60	6105.13	19,009.34	6611.98	2797.29	1058.58	315.64	136.01

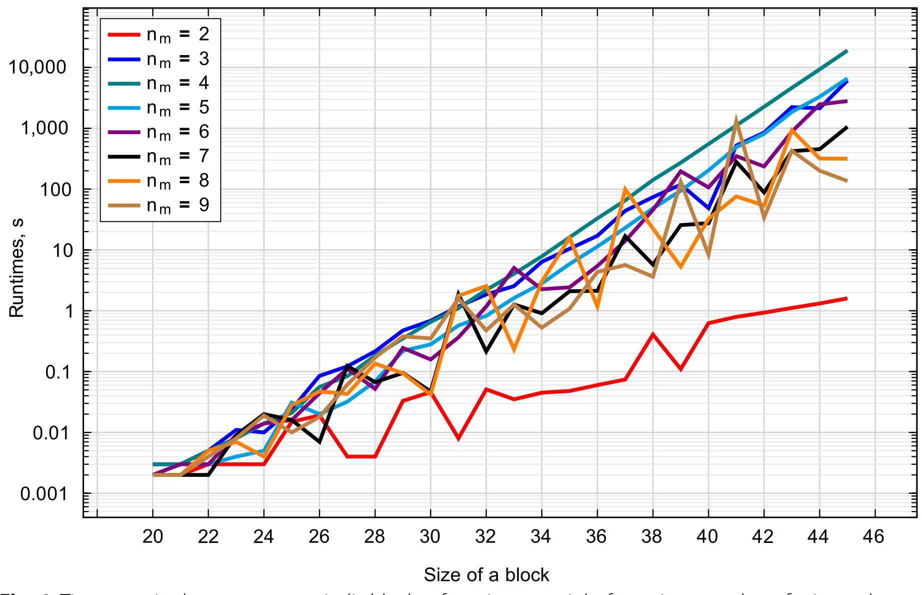


Fig. 6 Times required to generate periodic blocks of maximum weight for a given number of mismatches n_m and a known block size

of maximum weight. The detailed procedure is shown in Algorithm 2. We assume that we already have files with periodic blocks found for a set of lengths, from T_{min} to T_{max}, and valid for at most n_m mismatches. After application of Algorithm 2 we get a list Ω of n_{sol} periodic seeds of maximum weight. Note that for the same weight w and the number of mismatches n_m we may have seeds made of periodic blocks of different length/weight. Time required to run Algorithm 2 is the sum of times to process all blocks available for given values of n_m and T. While the number of valid blocks is different for a pair of n_m and T (from single entries to a million), it takes at most 5 s on Intel's i5-9600K CPU to find the best seed for a given length of a read.

Algorithm 2 Forming periodic spaced seeds of maximum weight for a given number of mismatches n_m

```
Require: $n_r$                                                    ▷ read's length
Require: $T_{\min}$, $T_{\max}$                  ▷ minimum/maximum size of valid blocks
 1: $w \leftarrow -1$                                              ▷ maximum weight
 2: $n_{sol} \leftarrow 0$                  ▷ number of solutions, $\Omega_i$, $i = 1, \ldots, n_{sol}$
 3: for $T \leftarrow T_{\min}, T_{\max}$ do
 4:     $n_s \leftarrow n_r - T + 1$                                ▷ seed's length
 5:     $n_b \leftarrow n_s/T$, $n_d \leftarrow n_s\%T$         ▷ integer division, remainder
 6:     if $n_b < 1$ then continue end if
 7:     if $n_d = 0$ then $n_d \leftarrow T$, $n_b \leftarrow n_b - 1$ end if
 8:     #1 $\leftarrow T$
 9:     open input file validBlock_#1.txt
10:     $K \leftarrow$ number of rows in the file
11:     for $j \leftarrow 1, K$ do
12:         read string $s_j$         ▷ each string is of length $T$, indices from 0 to $T - 1$
13:         $n_1 \leftarrow countOne(s_j, 0, T - 1)$   ▷ count number of 1-elements of substring
14:         for $q \leftarrow 1, T$ do
15:             for $p \leftarrow 0, T - 1$ do                     ▷ perform cyclic shift
16:                 $u[p] \leftarrow s_j[(q + p)\%T]$
17:             end for
18:             if $(u[0] = 0 \;||\; u[n_d - 1] = 0)$ then continue end if
19:             $n_2 \leftarrow countOne(u, 0, n_d - 1)$
20:             $w_{loc} \leftarrow n_b \cdot n_1 + n_2$
21:             if $w_{loc} < w$ then continue end if
22:             if $w_{loc} > w$ then
23:                 $w \leftarrow w_{loc}$, $n_{sol} \leftarrow 0$               ▷ reset all solutions
24:             end if
25:             $v \leftarrow$ concatenate $n_b$ whole strings $u$ and "remainder" of length $n_d$
26:             $n_{sol} \leftarrow n_{sol} + 1$, $\Omega_{n_{sol}} \leftarrow v$
27:         end for
28:     end for
29:     close input file validBlock_#1.txt
30: end for
```

Results

Periodic seeds and their properties

We have written a code to generate the best periodic blocks of size T for a given number n_m of mismatches. A user can specify an initial number n_1 of 1-elements in those blocks. If no valid blocks are found for a given n_1, then the value of n_1 is incremented by one and the procedure restarts until at least one valid block is found. The code was applied for n_m values from 2 to 9, and for T values from 10 to 50 (and from 10 to 70 for $n_m = 2$). Maximum weight n_1 and density of best blocks (defined as n_1/T) are shown in Table 4.

Note that while values n_1 and ρ tend to increase with the block size, they are not monotonic in general. See examples for $n_m = 5$ and $n_m = 8$ in Fig. 7 (more figures are in Additional file 1).

When all maximum-weight blocks are found for a given range of T, we may find periodic spaced seeds of the maximum weight for any given value of n_r. Of course, there may be cases of seeds composed of periodic blocks of different sizes. For example, if the length of reads is $n_r = 35$, the maximum weight of possible seeds is $w = 17$ and there are seeds formed for blocks of sizes 7, 10, 11, 13, e.g.

Table 4 Maximum weight n_1 and density ρ of periodic blocks (period T, number of mismatches n_m)

T	$n_m = 2$		$n_m = 3$		$n_m = 4$		$n_m = 5$		$n_m = 6$		$n_m = 7$		$n_m = 8$	
	n_1	$\rho, \%$	n_1	$\rho, \%$	n_1	$\rho, \%$	n_1	$\rho, \%$	n_1	$\rho, \%$	n_1	$\rho, \%$	n_1	$\rho, \%$
10	6	60.0	4	40.0	3	30.0	2	20.0	1	10.0	1	10.0	1	10.0
11	7	63.6	5	45.5	3	27.3	2	18.2	1	09.1	1	9.1	1	9.1
12	8	66.7	5	41.7	3	25.0	3	25.0	2	16.7	2	16.7	1	8.3
13	9	69.2	6	46.2	4	30.8	3	23.1	2	15.4	1	7.7	1	7.7
14	9	64.3	6	42.9	4	28.6	4	28.6	3	21.4	2	14.3	1	7.1
15	10	66.7	8	53.3	5	33.3	4	26.7	2	13.3	2	13.3	2	13.3
16	11	68.8	8	50.0	5	31.3	4	25.0	3	18.8	3	18.8	1	6.3
17	12	70.6	8	47.1	6	35.3	4	23.5	3	17.6	2	11.8	2	11.8
18	13	72.2	9	50.0	6	33.3	5	27.8	3	16.7	3	16.7	3	16.7
19	14	73.7	10	52.6	7	36.8	5	26.3	4	21.1	3	15.8	2	10.5
20	14	70.0	10	50.0	8	40.0	6	30.0	4	20.0	4	20.0	3	15.0
21	16	76.2	11	52.4	8	38.1	6	28.6	5	23.8	4	19.0	4	19.0
22	16	72.7	12	54.5	8	36.4	7	31.8	5	22.7	5	22.7	3	13.6
23	17	73.9	12	52.2	9	39.1	7	30.4	5	21.7	4	17.4	3	13.0
24	18	75.0	14	58.3	9	37.5	8	33.3	5	20.8	5	20.8	4	16.7
25	19	76.0	14	56.0	10	40.0	7	28.0	6	24.0	5	20.0	4	16.0
26	20	76.9	14	53.8	10	38.5	9	34.6	6	23.1	6	23.1	4	15.4
27	21	77.8	15	55.6	11	40.7	9	33.3	6	22.2	5	18.5	5	18.5
28	22	78.6	16	57.1	11	39.3	9	32.1	8	28.6	6	21.4	4	14.3
29	22	75.9	16	55.2	12	41.4	9	31.0	7	24.1	6	20.7	5	17.2
30	23	76.7	17	56.7	13	43.3	10	33.3	8	26.7	8	26.7	6	20.0
31	25	80.6	18	58.1	16	51.6	10	32.3	8	25.8	6	19.4	5	16.1
32	25	78.1	19	59.4	14	43.8	11	34.4	8	25.0	8	25.0	5	15.6
33	26	78.8	20	60.6	14	42.4	11	33.3	8	24.2	7	21.2	7	21.2
34	27	79.4	20	58.8	15	44.1	12	35.3	9	26.5	8	23.5	6	17.6
35	28	80.0	21	60.0	15	42.9	12	34.3	10	28.6	8	22.9	6	17.1
36	29	80.6	22	61.1	16	44.4	13	36.1	10	27.8	9	25.0	8	22.2
37	30	81.1	22	59.5	17	45.9	13	35.1	10	27.0	8	21.6	6	16.2
38	30	78.9	23	60.5	18	47.4	14	36.8	10	26.3	10	26.3	7	18.4
39	32	82.1	24	61.5	18	46.2	14	35.9	10	25.6	9	23.1	9	23.1
40	32	80.0	27	67.5	18	45.0	14	35.0	11	27.5	10	25.0	8	20.0
41	33	80.5	25	61.0	19	46.3	14	34.1	11	26.8	9	22.0	8	19.5
42	34	81.0	26	61.9	19	45.2	16	38.1	13	31.0	11	26.2	10	23.8
43	35	81.4	26	60.5	20	46.5	15	34.9	12	27.9	10	23.3	8	18.6
44	36	81.8	28	63.6	21	47.7	16	36.4	12	27.3	11	25.0	9	20.5
45	37	82.2	28	62.2	21	46.7	17	37.8	13	28.9	11	24.4	10	22.2
46	38	82.6	29	63.0	22	47.8	17	37.0	13	28.3	12	26.1	9	19.6
47	39	83.0	30	63.8	23	48.9	17	36.2	13	27.7	11	23.4	9	19.1
48	40	83.3	31	64.6	23	47.9	18	37.5	14	29.2	14	29.2	10	20.8
49	41	83.7	31	63.3	26	53.1	18	36.7	15	30.6	12	24.5	10	20.4
50	42	84.0	32	64.0	24	48.0	19	38.0	15	30.0	14	28.0	10	20.0

- $T = 7, n_b = 4, n_d = 1, b = 1011100$, $s = 10111001011100101110010111001$;
- $T = 10, n_b = 2, n_d = 6, b = 1101111000$, $s = 11011110001101111000110111$;
- $T = 11, n_b = 2, n_d = 3, b = 11111000110$, $s = 1111100011011111000110111$;
- $T = 13, n_b = 1, n_d = 10, b = 1011111011100$, $s = 10111110111001011111011$.

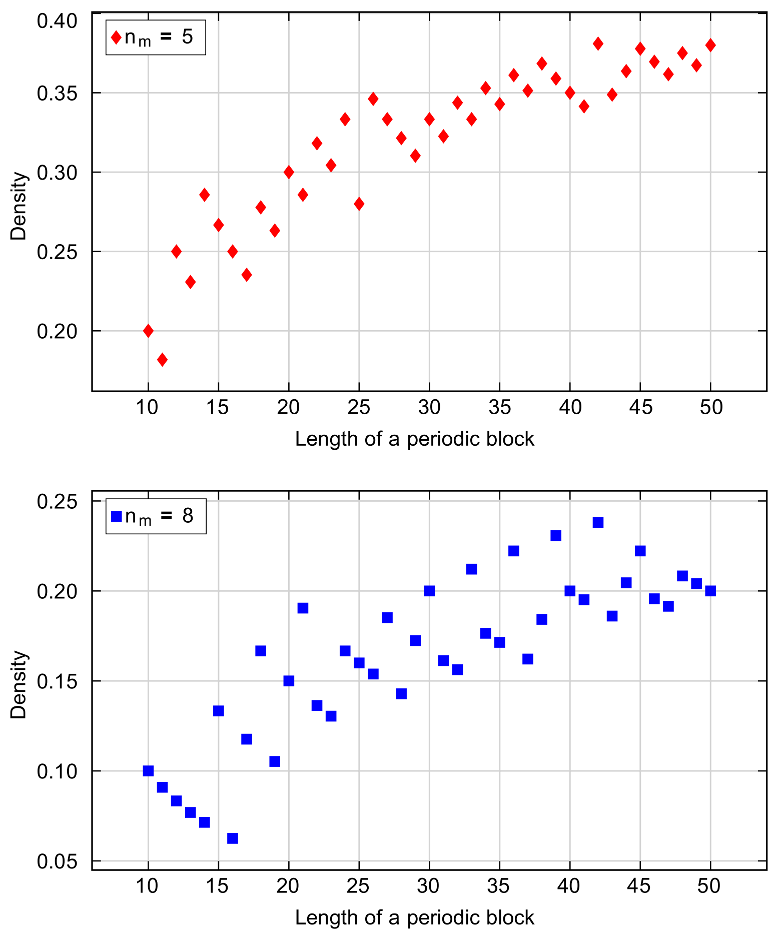
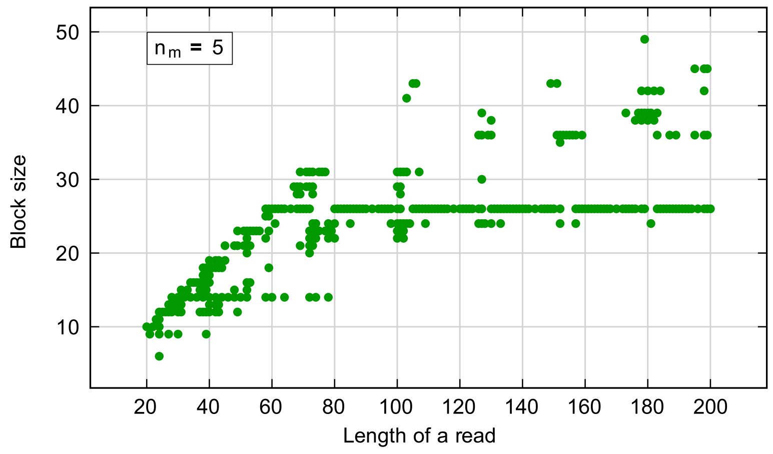


Fig. 7 Maximum numbers of 1-elements per length of periodic blocks, $n_m = 5$ and $n_m = 8$

For a given length n_r of reads, we plot sizes of corresponding periodic blocks, see Fig. 8. We see that sizes of best blocks tend to work in a specific range of n_r values. Best blocks are often those having peak density values. For example, if $n_m = 8$, the most frequent block sizes are $T = 18, 21, 33, 36, 39$ (Fig. 8), we see peak density values for them in Fig. 7. Note that values of T also increase with the length of reads. So, while we can find seeds for any n_r, they may not be the densest as there we have not generated periodic blocks for large values of T.

The final goal is to set the weight w of seeds, then find the minimum length of reads that have valid seeds of this weight and choose the longest seeds when several seeds are available. In Table 5, we present examples of these seeds for weights w (multiples of 8). As seeds may be very long, we show only periodic blocks, number n_b of these blocks and values of "remainders" n_d; see Tables 6 and 7. The complete list of spaced seeds found for various values of n_r and w ($n_r \le 400$, $w \le 320$) is in Additional file 1. Users can generate their seeds using codes (https://github.com/vtman/PerFSeeB).

We plot weights of the best seeds as a function of reads' length in Fig. 9. Based on lemmas in [32], it is possible to show that for a contiguous seed of weight/length q, the minimum length of reads when the seeds are valid is $K = q(n_m + 1)$, so the ratio is $r = q/K \to r_\infty = \frac{1}{n_m+1}$ when $q \to \infty$. The corresponding values of r_∞ are also shown in



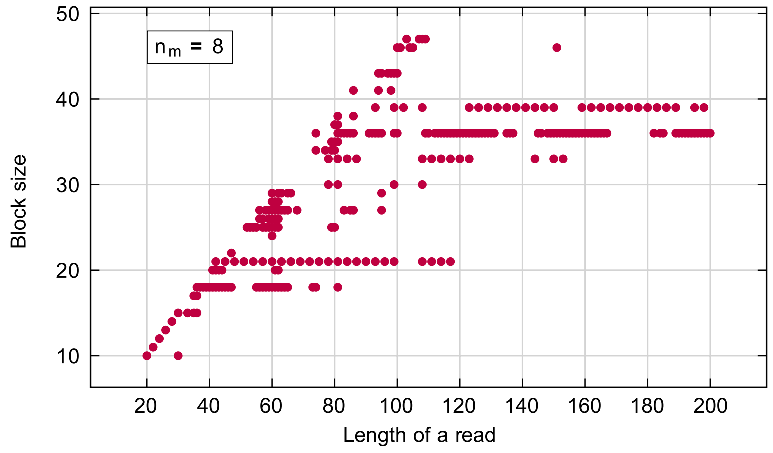


Fig. 8 Sizes of best periodic blocks for a given length of reads

Fig. 9 (dashed lines). Depending on values of n_m and n_r, the spaced seeds are denser (per read's length) compared to contiguous seeds by 20–68% ($n_r = 50$), 40–95% ($n_r = 100$), 60–113% ($n_r = 150$), 70–128% ($n_r = 200$).

Other seeds

To compare the quality of spaced seeds generated with the PerFSeeB approach, we present a list of the most popular seeds in Tables 8 and 9. These seeds are generated for given sensitivity levels. When there were several seeds presented in a paper, we chose a seed obtained for the highest sensitivity levels. We may see that the seeds are usually relatively short and of smaller weight. Unlike the PerFSeeB approach, the other algorithms did not usually put any restrictions on the number of mismatches. Several seeds were generated for a multiple-seed approach.

Comparison of seeds

To estimate performance characteristics, we develop software tools for real reference sequences and reads. For a reference sequence, we choose the T2T data [30]. Then, we randomly selected a relatively small dataset (ERR263486 [39], part of the 1000 Genomes Projects [40]): it is a paired-end data, and we use only the first dataset, ERR263486_1.

Table 5 Various longest periodic spaced seeds of the maximum weight for a given number of mismatches n_m and weight w

n_m	w	n_r	**Longest spaced seed**
2	16	32	1011100101110010111001011 1
2	20	39	10111001011100101110010111001 0111
2	24	44	111111001101011111100110101 11111
2	28	51	110111110001101111100011011111000110 11111
2	32	56	111110111001011111011100101111101110010 11111
2	36	62	10111110111001011111011100101111101110010111110111
2	40	68	1111101110010111110111001011111011100101111101110010 1111
2	44	73	111110111001011111011100101111101110010111110111001011111 0111
3	16	41	10011010111100010011010 1111
3	20	48	111100010011010111100010011010 1111
3	24	56	100110101111000100110101111000100110101111
3	28	63	1111000100110101111000100110101111000100110101111
3	32	71	1001101011110001001101011110001001101011110001001101 01111
3	36	78	111100010011010111100010011010111100010011010111100010011010 1111
4	16	54	11110010000000100011110010000000100 01111
4	20	64	1101110100000001000011011101000000010000110 11101
4	24	70	1110111110010011000010110101000111011111
4	28	79	1100111110001101110101000010010110011111000110111
4	32	88	1001011001111100011011101010000100101100111110001101110101
4	36	96	11111000110111010100001001011001111100011011101010000100101100 1111
5	16	62	1110011010100000001001000000111001 10101
5	20	76	110001101011111000000000000000000110001101011111
5	24	86	111001101010000000100100000011100110101000000010010000001110 01101
6	16	75	1001101011110000000000000000000000100110101111
6	20	91	1000011100000001000000110000101000011100000001000000110000101000 0111
6	24	102	11001101010111111000000000000000000000000000110011010101 11111
7	16	82	100000101000100010101010000000010000010100010001010101
7	20	96	1010101000000010000010100010001010101000000010000010100010001010101

Table 6 Blocks for spaced seeds ($n_m = 2$)

w	n_r	**Periodic block**	n_b	n_d
16	32	1011100	3	5
24	44	1111110011010	2	6
32	56	1111101110010	3	5
40	68	1111101110010	4	4
48	80	1111101110010	5	3
56	91	1110110111111111000	3	16
64	102	1111111110001110110	4	8
72	112	11111111101011100110	4	8
80	123	11011111111101011100	4	19
88	133	11111111101011100110	5	8
96	144	11011111111101011100	5	19

Table 7 Blocks for spaced seeds ($n_m = 6$)

w	n_r	Periodic block	n_b	n_d
16	75	10011010111100000000000000000000	1	12
24	102	1100110101011111100000000000000000000000000	1	17
32	133	1000001100001010000111000000	3	22
40	161	1000001100001010000111000000	4	22
48	188	1100110101011111100000000000000000000000000	3	17
56	213	111100000010000011000010100000110001001000	4	4
64	239	110000011000010100001110000111000000100000	4	30
72	266	100001110000111000000100000110000011000010	5	15
80	295	110000011000010100001110000111000000100000	6	2
88	316	110000101000011100001110000001000001100000	6	23
96	343	100011110000001000001100001010000011000100	7	8

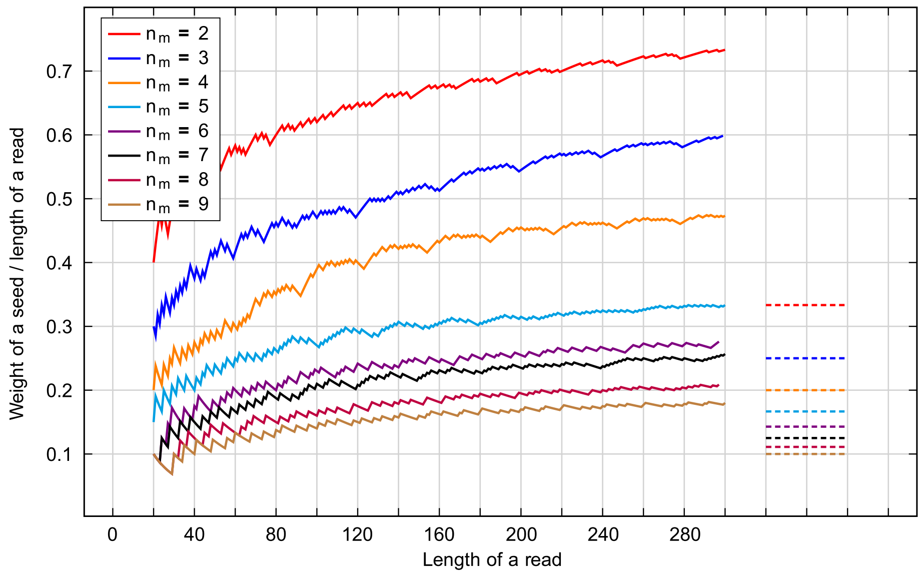


Fig. 9 Weights of the best spaced seeds per reads' lengths (solid lines). Maximum ratios for contiguous seeds (dashed lines)

1. There are 1734496 reads of length 100; we removed reads containing `N` symbols. Thus we get 1731924 reads. For each read, we also create its counterpart, i.e., flip the sequence and change `A` $\leftrightarrow$ `T`, `C` $\leftrightarrow$ `G`, so for `ATCCCAAAGTGCTTT` we generate `AAA GCACTTTGGGAT`. Therefore we use 3463848 sequences.
2. For a given seed, we generate the corresponding library of records ("key", "value") where "key" is the position within the reference sequence and sort it by "values".
3. For a chosen seed of length n_s we create $(101 - n_s)$ "values" for each read. For each "value" we find the list of all "keys" in the library (if the "value" is present). Knowing the relative position of the seed to the read, we use the found seed to create a new list. All elements of this list are possible candidate positions at the start of the read. As we have at most $(101 - n_s)$ such lists, we merge them into one list and remove all duplicated entries. We count the total number of possible candidate positions of the

Table 8 Examples of most popular spaced seeds

[33]	**Diamond**	[34]	**BFAST**
D_1	111101011101111	B_1	1111111111111111111111
D_2	111011001100101111	B_2	1111010010110110101110010110111011
D_3	1111001001010001001111	B_3	1111101101110111101111111
D_4	111100101000010010010111	B_4	10111101011010010110000110100011111111
		B_5	1111101110111010100101011011111
[8]	PatternHunter	B_6	101110011010011001001111010100010111111
P_1	111010010100110111	B_7	1111111001010010001011111101110111
[17]	PatternHunter II	B_8	111101101011011001100000101101001011101
P_2	111100110010100001011	B_9	111101011100101000101011010101111111
P_3	110100001100010101111	B_{10}	1111011010001000110101100101100110100111
P_4	1110111010001111		
		[35]	rasbhari
[36]	Quadratic Residues	R_1	1111011110011010111110101011011
Q_1	101100001	R_2	1110101011101100110100111111111
Q_2	10110000101011111001	R_3	1111110101101011100111011001111
[37]			
S_1	11111111110001110110010010011101001110001010010100001010011000010111100000011		
S_2	1111011111101100111100011010110011101011000000111010001101001010011110011		
S_3	1110110101001101011001001001010110001		
S_4	111111111111111100011111110000110011000111001111000011100000000111101110000011		
S_5	11000001010111101001000000001101111111111000110000101111100101110111010110001 01		
S_6	1110001000011110110101100000100010000111010110110110111101011011000100100101011		

read with respect to the reference sequence. Those candidate positions may be used for proper read alignment. Ideally, we do not want to miss any position that may provide us with proper alignment (accounting for known restrictions), but at the same time, we what to keep the number of these positions to a minimum to increase the performance of a sequence alignment code.

4. Our goal is to see how seeds perform under our restrictions (when only at most a given number of SNPs is possible). We check all candidate positions for each read and its counterpart and find the maximum number of matches. If two seeds are to be compared, then the best seed should provide us with more matches. Of course, there may be several positions when the number of matches attains its maximum value (possible for the given seed). If two seeds provide the same maximum number of matches for a read, then the best seed should provide us with a longer list of the best candidate positions.

To assist a reader, we developed software tools to deal with users' seeds and provide them with lists of all and best candidate positions and information about the total number of items in the lists and the maximum number of matches attained. The software is also available at (https://github.com/vtman/PerFSeeB):

1. **fna2bin** and **ref2m128**—convert input FNA file for the T2T reference data into binary files;
2. **fastq2bin** and **bin2m128**—convert FASTQ files into binary files;

Table 9 Spaces seeds for MegaBLAST [38]

M_1	1111101101011001101100011110010101011101111
M_2^1	111111010111011110111011011011101111
M_2^2	11111101110011001101001011010111010111111
M_4^1	1111111011111011110111111011111111
M_4^2	11111101101011011100100011111001010111111
M_4^3	11110111010011001010111101101110011110111
M_4^4	11111010111000111011011010001011111101111
M_8^1	11111010111011101111011110111011111
M_8^2	111001111110101111011001011111101110111
M_8^3	1111011111011111000110010110101111101111
M_8^4	1111111001011110110011101001110110101111
M_8^5	11110110111010110110101011100011011011111
M_8^6	11111011011100101000111111011000101111111
M_8^7	11110101100110011111010111001001111011111
M_8^8	11101110100100110111110001111110101101111
M_{16}^1	11111101111011111011111011111111
M_{16}^2	111010111111011011111111100111110111
M_{16}^3	111110111110111110001101110111011111
M_{16}^4	111101110111111011110100011110111111
M_{16}^5	1111111000110111101110111101111110111
M_{16}^6	1111110111110101011011011011100111111
M_{16}^7	11110111011001001111011111110101011111
M_{16}^8	111111011011110010011110101010011111111
M_{16}^9	111110111100101110010111101101101101111
M_{16}^{10}	1111100111011101111101001110010010111111
M_{16}^{11}	1111011011011111110100011010110101101111
M_{16}^{12}	11110101101110110000111001110110111101111
M_{16}^{13}	11111110010010111111011101001001110101111
M_{16}^{14}	11110111010100011101101000111111100111111
M_{16}^{15}	11111011000110100110001011111111011011111
M_{16}^{16}	11111011110101100011101110000111010111111

3. **createList**—create files for an unsorted records for the given reference sequence and seed;
4. **sortList**—sort the records in ascending order for “values”;
5. **searchPositions**—find all candidate positions of reads by using the library of records (we also get a total number of “values” for each read and all possible and unique positions of the seed);
6. **countMatch**—for each read, we check all candidate positions in the reference sequence and report the maximum number of matching symbols and all positions where the maximum is attained.

We create several seeds using PerFSeeB tools. The output files for the ERR263486 dataset can be found in https://github.com/vtman/PerFSeeB. For reads’ length $n_r = 100$, we use seeds listed in Additional file 1 (see Table 10). We use for comparison seeds N_2, N_3,

N_4 from the table and seeds S_1 and $\overline{S_3}$ from Table 8 ($\overline{S_3}$ is for multiple seeds S_1, S_2, S_4, S_5, S_6; S_3 is excluded as a seed of low weight). We may see in Table 11 that the total number of reads attaining a given number of matches is the same for all seeds and 98, 99, 100 matches (or, equivalently, for the number of mismatches $n_m = 2, 1, 0$). However, if we want to find reads fully matching the reference sequence at some positions, then to do this, we need 1,140,090,415 candidate positions to be checked for S_1, and 1,980,598,089 for five multiple seeds $\overline{S_3}$. At the same time seed N_2 requires only 306,980,283 candidate positions (3.7 and 6.5 times less compared to S_1 and $\overline{S_3}$). We also see that the average ratio of all to best candidate positions is 10 and 22 times smaller for N_2. And reads with two or fewer mismatches are 95% of all reads in the dataset.

Note that we may also apply new seeds as multiple seeds. For example, we may use seed N_2 and find all reads with at least 98 matches. Then for the remaining reads, we may use another seed. Let the total number of candidate positions to be processed for seed N_2 be β. If we consider seed N_2 followed by seed N_3 (denote it as $N_2 \cup N_3$), then we find all reads that have at least 97 matches, and in this case, we should process $\approx 1.026\beta$. However, if we use N_3 alone, then we require 2.056β candidate positions to be processed (see numbers in Table 11). For other combinations with N_2 we get 1.133β for $N_2 \cup N_4$, 2.099β for $N_2 \cup N_5$, 2.570β for $N_2 \cup N_6$, 5.029β for $N_2 \cup N_7$, 7.871β for $N_2 \cup N_8$. However, if we use all seeds one by one, i.e. $N_2 \cup N_3 \cup \ldots \cup N_8$, we get 4.434β.

The proposed seeds or their combinations allow us to process a relatively small number of candidate positions and find all reads having not more than a given number of mismatches. In Table 12, we provide a number of candidate positions to be processed for each seed. We see that for seed N_2 applied to the real-life data, we need around 208 positions to be checked. All other seeds require significantly higher numbers of positions to be processed. The other good seeds are $S_1, \ldots, S_6$, but they have been discussed above.

Our goal is to have full-sensitivity seeds. Therefore we aim to check how the other seeds work under such requirements (full sensitivity for a given maximum number n_m of mismatches). For each seed, we vary the length n_r of reads and check if the seed is

Table 10 Seeds generated with PerFSeeB for $n_r = 100$ (seeds' weight is in the second column)

N_2	62	11111011100101111101110010111110111001011111011100101111101110010
		1111101
		→110010111110111
N_3	47	11010111100010011010111100010011010111100010011010111100010011010
		1111000
		→100110101111
N_4	39	11101111100100110000101101010001110111110010011000010110101000111
		01111
N_5	27	10001001011000111010000000100010010110001110100000001000100101100
		011101
N_6	23	11000010100001110000001000001100001010000111000000100000110000101
		0000111
N_7	21	10001010101000000010000010100010001010101000000010000010100010001
		010101
N_8	16	10000010010010000000010000010010010000000010000010010010000000010
		0000100
		→1001

Table 11 For each read and its counterpart, we find the number of unique candidate positions found for given seeds and the number of positions having the maximum number of matches with the read

	S_1	$\overline{S_3}$	N_2	N_3	N_4
Percentage of reads having at least given number of matches					
95	96.49413	96.55955	95.88331	96.41693	96.54956
96	96.23315	96.26387	95.77562	96.20613	96.26404
97	95.81032	95.81777	95.56522	95.81777	95.81777
98	94.98875	94.98875	94.98875	94.98875	94.98875
99	92.66636	92.66636	92.66636	92.66636	92.66636
100	81.17036	81.17036	81.17036	81.17036	81.17036
Total number of positions having the maximum number of matches					
95	24,673	30,124	5,743	18,078	29,184
96	88,548	95,152	28,715	76,571	96,154
97	271,116	272,204	165,624	272,204	272,204
98	1,072,182	1,072,182	1,072,182	1,072,182	1,072,182
99	6,505,597	6,505,597	6,505,597	6,505,597	6,505,597
100	52,432,760	52,432,760	52,432,760	52,432,760	52,432,760
Total number of candidate positions to be checked					
95	2,999,321	7,694,425	62,080	534,856	5,075,260
96	6,484,344	15,560,907	183,070	1,785,944	10,215,791
97	16,738,989	35,342,085	975,884	6,106,930	23,375,078
98	49,957,296	94,777,228	6,591,836	22,966,268	66,288,831
99	206,562,418	374,668,653	42,466,387	101,753,217	268,804,889
100	1,140,090,415	1,980,598,089	306,980,283	593,471,736	1,467,713,805
Averaged ratio of candidate positions to best positions					
95	334.88	790.87	10.37	50.87	535.79
96	414.02	1039.40	7.57	100.63	677.69
97	493.92	1164.11	14.72	115.13	744.46
98	486.19	1087.47	22.87	137.82	704.10
99	465.90	1001.87	31.37	140.65	662.25
100	412.66	848.60	38.46	137.69	575.35

(a) Percentage of reads attaining at least a given number of matches (higher values are better). (b) The total number of positions attaining the maximum number of matches (we add up numbers for all reads, higher values are better). (c) The total number of candidate positions to be checked (smaller values are better). (d) For each read attaining the given number of matches, we find the ratio of all candidate positions to the number of best positions (averaged; smaller values are better)

valid. At the same time, we may use the found size n_r and design seeds according to the PerFSeeB procedure. Their weights are also found and compared to the weights of the original seeds. The weights are usually 30–40% higher compared to the weights of the original seeds. See Table 13. We may see that seeds generated directly with the extension procedure (usually $n_r < 45$) and seeds developed with the PerFSeeB approach (they are all called "best" seeds in Table 13) are valid for shorter reads: by approximately 15% for $w \leq 12$, by 20% for $w = 20, 22$ and by 32% for $w = 39$–46.

Table 12 The average number of candidate positions required to be checked for each read. Index "all" is used when all seeds in a group are used, and the corresponding lists of candidate positions are merged and duplicate positions removed from the final list

B_1	33030	B_2	35709	B_3	29874	B_4	28531	B_5	41961	B_6	34353
B_7	30830	B_8	27054	B_9	27023	B_{10}	33030	B_{all}	98503		
M_1	9245	M_2^1	11672	M_2^2	9657	M_2^{all}	16102				
M_4^1	13509	M_4^2	9742	M_4^3	9690	M_4^4	9665	M_4^{all}	23645		
M_8^1	12351	M_8^2	10968	M_8^3	10194	M_8^4	10456	M_8^5	9758	M_8^6	9669
M_8^7	9928	M_8^8	9524	M_8^{all}	29170						
M_{16}^1	13467	M_{16}^2	12357	M_{16}^3	11835	M_{16}^4	11832	M_{16}^5	11812	M_{16}^6	11247
M_{16}^7	10754	M_{16}^8	10597	M_{16}^9	10619	M_{16}^{10}	10502	M_{16}^{11}	10168	M_{16}^{12}	9762
M_{16}^{13}	9753	M_{16}^{14}	9751	M_{16}^{15}	9723	M_{16}^{16}	9724	M_{16}^{all}	39922		
N_2	208	N_3	423	N_4	1062	N_5	4283	N_6	6888	N_7	11529
N_8	22254	R_1	36532	R_2	35816	R_3	36393	R_{all}	59239	S_3	37046
S_1	825	S_2	674	S_4	504	S_5	624	S_6	706	$\overline{S_3}$	1451

Discussion

The PerFSeeB approach allows us to generate long, high-weight spaced seeds for a maximum number of mismatches. The analysis of performance for the read alignment algorithm was based on the simplistic approach: a user generates a list of possible "values" and finds all candidate positions mapped with these "values" in the reference sequence. For each position of a seed to a read, we get "value" and a list of positions in the reference sequence. Thus for a read of length n_r and a seed of size n_s, we get $(n_r - n_s + 1)$ "values" and the same number of lists. If a read can be perfectly positioned in the reference sequence (100% match), then all $(n_r - n_s + 1)$ lists of candidate positions should have all the same positions. Therefore we may start dealing with the intersection of all lists. Since each "value" may provide us with lists of different lengths (some "values" may have thousands of "keys"), intersecting lists may significantly reduce the number of candidate positions. If some "values" provide us with empty lists, this may help us to identify possible positions of a mismatch by analysing the location of "do not care" symbols within the seed and how lists of candidate positions vary with starting positions of the seed.

Modern data collection procedures have paired-end reads such that the distance between reads can be roughly estimated, e.g. within a thousand symbols. Therefore, one can analyse lists obtained for two reads within a pair to further reduce the number of candidate positions. This may be extremely helpful when one of the reads has insertions and deletions, as we may start working with a read not containing indels. Of course, candidate positions for relatively long reads with a small number of indels can also be found with the proposed seeds. However, proper seed design dealing with indels is still needed, so we must catch all of them. For example, seeds with relatively big chunks of "do not care" symbols inside may be processed with reads split into several parts and each part shifted by a couple of characters. If the number of reads is big, one may also try to align reads with respect to each other by comparing starting/ending chunks.

Of course, one can use the PerFSeeB approach for long reads. For example, seeds generated with the PerFSeeB approach and listed in Additional file 1 can be used for reads

Table 13 Minimum read lengths are required to achieve full sensitivity for a given number n_m of mismatches

Seed	w	$n_m = 2$	$n_m = 3$	$n_m = 4$	$n_m = 5$	$n_m = 6$	$n_m = 7$
Q_1	4	14 (5)	16 (4)	21 (5)	24 (4)	28 (4)	30 (4)
best		**11**	**14**	**17**	**20**	**23**	**26**
Q_2	10	28 (13)	37 (14)	44 (12)	52	58	65
best		**22**	**29**	**37**	**44**	**52**	**58**
P_1	11	27 (12)	40 (15)	46 (13)	51	62	68
P_2		32 (16)	37 (14)	50 (14)	58	63	72
P_3		31 (15)	37 (14)	48 (14)	55	62	70
P_4		28 (13)	38 (15)	44 (12)	52	60	68
best		**23**	**31**	**40**	**46**	**56**	**62**
D_1	12	33 (16)	38 (15)	51 (15)	56	69	74
D_2		31 (15)	41 (16)	49 (14)	58	67	76
D_3		30 (15)	39 (15)	55 (16)	60	66	77
D_4		33 (16)	40 (15)	56 (17)	63 (16)	68 (17)	76
best		**25**	**33**	**43**	**50**	**59**	**66**
S_3	20	51 (28)	76 (34)	85 (31)	95 (26)	114 (27)	
best		**39**	**48**	**64**	**76**	**91**	
B_1	22	66 (38)	88 (40)	110 (44)	132 (39)	154 (38)	
B_2		49 (27)	68 (31)	80 (28)	94 (26)	111 (25)	
B_3		55 (31)	82 (38)	90 (32)	102 (28)	127 (31)	
B_4		51 (28)	70 (31)	88 (32)	97 (27)	112 (26)	
B_5		54 (30)	65 (29)	82 (29)	94 (26)	110 (25)	
B_6		49 (27)	76 (34)	84 (30)	97 (27)	119 (28)	
B_7		54 (30)	72 (32)	89 (32)	100 (27)	116 (27)	
B_8		58 (33)	68 (31)	91 (32)	101 (27)	117 (27)	
B_9		54 (30)	69 (31)	90 (32)	97 (27)	111 (25)	
B_{10}		55 (31)	73 (32)	88 (32)	102 (28)	113 (26)	
R_1		54 (30)	68 (31)	85 (31)	99 (27)	112 (26)	
R_2		54 (30)	74 (32)	86 (31)	98 (27)	117 (27)	
R_3		50 (27)	70 (31)	81 (29)	97 (27)	114 (27)	
best		**42**	**52**	**68**	**82**	**97**	
S_1	39	98 (61)	121 (58)	160 (69)	176 (54)		
best		**67**	**83**	**100**	**132**		
S_6	40	107 (68)	131 (65)	163 (72)	179 (54)		
best		**68**	**86**	**101**	**134**		
S_5	42	100 (62)	149 (77)	172 (76)	189 (59)		
best		**70**	**91**	**105**	**138**		
S_2	43	96 (60)	123 (60)	163 (72)	180 (55)		
best		**72**	**92**	**107**	**140**		
S_4	46	115 (74)	151 (78)	187 (82)	223 (71)		
best		**78**	**97**	**114**	**153**		

For comparison with offered seeds, we also provide numbers for the best seeds found with the PerFSeeB approach. The corresponding weights are in parentheses, and the shortest possible reads' length for w and n_m are in bold for each "best" row

up to 500 symbols. However, high-weight seeds also require almost proportionally larger storage. Therefore dealing with shorter chunks of long reads may be more practical as this may also help to avoid possible insertions/deletions.

Some ideas used in the paper have already been discussed for other projects. Lossless seeds were introduced in [9] and generalised in [24]. In [36] structure of seeds for $n_m = 2$ and 3 was studied in depth. In [15], the periodic spaced seeds were also discussed. However, full sensitivity seeds of maximum weight are considered for at most three mismatches. At the same time, seeds' weights were very small, e.g. the number of 1 s in a periodic block was at most five. For this paper, we designed an optimised publicly available code to generate all possible seeds continuously. We have extensively computed all possible periodic blocks of the maximum weight for a given size T of a block and the number of mismatches n_m. For example, for $n_m = 2$ and $T = 70$, we found all blocks of weight 60; for $n_m = 4$ and $T = 50$, we get blocks of weight 24. In [41], the authors presented a method for fast computation of optimal multiple-spaced seeds of weight 11. In addition, they introduce overlap complexity, a measure which correlates with sensitivity. In the PerFSeeB approach, we design single lossless seeds (but also showed that the consequent application of several seeds reduces the number of candidates). Meanwhile, our approach does not account for a common or prohibited small substring in reference sequences. This approach was proposed in [42]. It can, in principle, be combined with the PerFSeeB approach when seeds or periodic blocks of maximum (or near maximum) weights are generated and applied to reference sequences to have the number of "keys" in library of records more evenly distributed.

Conclusions

The PerFSeeB approach proposed in this paper is based on designing periodic blocks. When several mismatches are set, resulting spaced seeds are guaranteed to find all positions within a reference sequence. Each periodic seed consists of an integer number of periodic blocks and a "remainder" (a number of the first symbols of the block). The size of the periodic block is the difference between the read's and seed's lengths plus one. This relation is empirical and was observed for seeds generated by iterative extension procedure for reads of length less than 45.

Periodic blocks are found for the number mismatches n_m from 2 to 9 and block's length T up to 50 (or 70 for $n_m = 2$); they can in be accessed at https://github.com/vtman/PerFSeeB. Those blocks can be used to generate spaced seeds required for any given length of reads. The best periodic seeds are seeds of maximum possible weight since this helps us to reduce the number of candidate positions when we try to align reads to the reference sequence. If one can generate several best seeds, we choose seeds of maximum length because this helps us reduce the number of entries to be checked in the library found for the reference sequence.

The seeds found with the PerFSeeB approach might be of lower weight for long reads (>200) as we have to generate longer periodic blocks; nevertheless, they meet the requirements. Furthermore, codes used to create periodic blocks and final seeds are designed to account for SIMD instructions, can work in a multithreading environment and are publicly available at (https://github.com/vtman/PerFSeeB). While the authors did their best to minimise the number of candidate blocks to be validated, there may be trillions of them to be checked for periods T around 50.

The proposed approach allows us to check significantly fewer alignment positions for each read than other published seeds. This, in turn, can be exploited in sequence alignment algorithms to improve performance and ensure not to miss positions due to mismatches.

Supplementary Information

The online version contains supplementary material available at https://doi.org/10.1186/s12859-023-05517-4.

Additional file 1: Maximum density of best periodic blocks as a function of block's size and block sizes as a function of read's length (for 2 to 9 mismatches).

Acknowledgements
The authors would like to thank Dr. Laurent Noé for fruitful discussion of the ideas and his comments.

Author Contributions
VT and ST conceived the ideas for the study. VT wrote codes to generate the seeds. Both authors wrote the manuscript and approved its final version.

Funding
Not applicable.

Availability of data and materials
The codes to generate periodic blocks and seeds are publicly available at https://github.com/vtman/PerFSeeB.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 4 November 2021 Accepted: 2 October 2023
Published online: 24 October 2023

References

1. Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mol Biol. 1970;48(3):443–53. https://doi.org/10.1016/0022-2836(70)90057-4.
2. Smith TF, Waterman MS. Identification of common molecular subsequences. J Mol Biol. 1981;147(1):195–7. https://doi.org/10.1016/0022-2836(81)90087-5.
3. Durbin R, Eddy SR, Krogh A, Mitchison G. Biological sequence analysis: probabilistic models of proteins and nucleic acids. Cambridge: Cambridge University Press; 1998. https://doi.org/10.1017/CBO9780511790492.
4. Head SR, Komori HK, LaMere SA, Whisenant T, Van Nieuwerburgh F, Salomon DR, Ordoukhanian P. Library construction for next-generation sequencing: overviews and challenges. Biotechniques. 2014;56(2):61–77. https://doi.org/10.2144/000114133.
5. Wang B, Kumar V, Olson A, Ware D. Reviving the transcriptome studies: an insight into the emergence of single-molecule transcriptome sequencing. Front Genet. 2019;10:384. https://doi.org/10.3389/fgene.2019.00384.
6. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. J Mol Biol. 1990;215(3):403–10. https://doi.org/10.1016/S0022-2836(05)80360-2.
7. Altschul SF, Madden TL, Schäffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res. 1997;25(17):3389–402. https://doi.org/10.1093/nar/25.17.3389.
8. Ma B, Tromp J, Li M. PatternHunter: faster and more sensitive homology search. Bioinformatics. 2002;18(3):440–5. https://doi.org/10.1093/bioinformatics/18.3.440.
9. Burkhardt S, Kärkkäinen J. Better filtering with gapped q-grams. In: Amir A, editor. Combinatorial pattern matching. Berlin: Springer; 2001. p. 73–85.
10. Choi KP, Zeng F, Zhang L. Good spaced seeds for homology search. Bioinformatics. 2004;20(7):1053–9. https://doi.org/10.1093/bioinformatics/bth037.
11. Brejová B, Brown DG, Vinař T. Vector seeds: an extension to spaced seeds. J Comput Syst Sci. 2005;70(3):364–80. https://doi.org/10.1016/j.jcss.2004.12.008.
12. Mak D, Gelfand Y, Benson G. Indel seeds for homology search. Bioinformatics. 2006;22(14):341–9. https://doi.org/10.1093/bioinformatics/btl263.
13. Csűrös M, Ma B. Rapid homology search with neighbor seeds. Algorithmica. 2007;48(2):187–202. https://doi.org/10.1007/s00453-007-0062-y.

14. Lin H, Zhang Z, Zhang MQ, Ma B, Li M. ZOOM! Zillions of oligos mapped. Bioinformatics. 2008;24(21):2431–7. https://doi.org/10.1093/bioinformatics/btn416.
15. Chen Y, Souaiaia T, Chen T. PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds. Bioinformatics. 2009;25(19):2514–21. https://doi.org/10.1093/bioinformatics/btp486.
16. Leimeister C-A, Boden M, Horwege S, Lindner S, Morgenstern B. Fast alignment-free sequence comparison using spaced-word frequencies. Bioinformatics. 2014;30(14):1991–9. https://doi.org/10.1093/bioinformatics/btu177.
17. Li M, Ma B, Kisman D, Tromp J. PatternHunter II: highly sensitive and fast homology search. J Bioinform Comput Biol. 2004;2(3):417–39. https://doi.org/10.1142/S0219720004000661.
18. Sun Y, Buhler J. Designing multiple simultaneous seeds for DNA similarity search. J Comput Biol. 2005;12(6):847–61. https://doi.org/10.1089/cmb.2005.12.847.
19. Brown DG. 6. A survey of seeding for sequence alignment. Hoboken: Wiley; 2008. p. 117–42. https://doi.org/10.1002/9780470253441.ch6.
20. Noé L. Spaced seeds. Accessed 30 October 2022, 2022.
21. What is discontiguous Mega BLAST? https://blast.ncbi.nlm.nih.gov/doc/blast-topics/discontmegablast.html. Accessed 30 July 2023.
22. Gambin A, Lasota S, Startek M, Sykulski M, Noé L, Kucherov G. Subset seed extension to protein BLAST. In: Proceedings of the International Conference on Bioinformatics Models, Methods and Algorithms (BIOSTEC 2011)—BIOINFORMATICS, SciTePress, Rome, Italy 2011;149–158. https://doi.org/10.5220/0003147601490158. INSTICC.
23. Noé L, Kucherov G. YASS: enhancing the sensitivity of DNA similarity search. Nucl Acids Res. 2005;33(suppl-2):540–3. https://doi.org/10.1093/nar/gki478.
24. Kucherov G, Noe L, Roytberg M. Multiseed lossless filtration. IEEE/ACM Trans Comput Biol Bioinf. 2005;2(1):51–61. https://doi.org/10.1109/TCBB.2005.12.
25. Hamming RW. Error detecting and error correcting codes. Bell Syst Tech J. 1950;29(2):147–60. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x.
26. Navarro G. A guided tour to approximate string matching. ACM Comput Surv. 2001;33(1):31–88. https://doi.org/10.1145/375360.375365.
27. Levenshtein VI. Bounds for codes ensuring error correction and synchronization. Probl Inf Transm. 1969;5:1–10.
28. Kruskal JB. An overview of sequence comparison: time warps, string edits, and macromolecules. SIAM Rev. 1983;25(2):201–37. https://doi.org/10.1137/1025045.
29. Apostolico A, Guerra C. The longest common subsequence problem revisited. Algorithmica. 1987;2(1):315–36. https://doi.org/10.1007/BF01840365.
30. T2T Consortium: T2T CHM13v2.0 Telomere-to-Telomere assembly of the CHM13 cell line. Accessed 30 October 2022, 2022.
31. Intel: Intel Intrinsics Guide. Accessed 30 October 2022, 2022.
32. Pevzner PA, Waterman MS. Multiple filtration and approximate pattern matching. Algorithmica. 1995;13(1):135–54. https://doi.org/10.1007/BF01188584.
33. Buchfink B, Xie C, Huson DH. Fast and sensitive protein alignment using DIAMOND. Nat Methods. 2015;12(1):59–60. https://doi.org/10.1038/nmeth.3176.
34. Homer N, Merriman B, Nelson SF. BFAST: an alignment tool for large scale genome resequencing. PLoS ONE. 2009;4(11):1–12. https://doi.org/10.1371/journal.pone.0007767.
35. Hahn L, Leimeister C-A, Ounit R, Lonardi S, Morgenstern B. rasbhari: optimizing spaced seeds for database searching, read mapping and alignment-free sequence comparison. PLoS Comput Biol. 2016;12(10):1–18. https://doi.org/10.1371/journal.pcbi.1005107.
36. Egidi L, Manzini G. Better spaced seeds using quadratic residues. J Comput Syst Sci. 2013;79(7):1144–55. https://doi.org/10.1016/j.jcss.2013.03.002.
37. Salmela L, Mukherjee K, Puglisi SJ, Muggli MD, Boucher C. Fast and accurate correction of optical mapping data via spaced seeds. Bioinformatics. 2019;36(3):682–9. https://doi.org/10.1093/bioinformatics/btz663.
38. Ilie S. Efficient computation of spaced seeds. BMC Res Notes. 2012;5(1):123–11237. https://doi.org/10.1186/1756-0500-5-123.
39. IGSR: Biosample 124940. Accessed 30 October 2022, 2013.
40. Fairley S, Lowy-Gallego E, Perry E, Flicek P. The International Genome Sample Resource (IGSR) collection of open human genomic variation resources. Nucleic Acids Res. 2019;48(D1):941–7. https://doi.org/10.1093/nar/gkz836.
41. Ilie L, Ilie S. Multiple spaced seeds for homology search. Bioinformatics. 2007;23(22):2969–77. https://doi.org/10.1093/bioinformatics/btm422.
42. Frith MC, Noé L, Kucherov G. Minimally overlapping words for sequence similarity search. Bioinformatics. 2020;36(22–23):5344–50. https://doi.org/10.1093/bioinformatics/btaa1054.