KFinger: Capturing Overlaps between Long Reads by Using Lyndon Fingerprints*

Paola Bonizzoni[1], Alessia Petescia[1], Yuri Pirola[1], Raffaella Rizzi[1], Rocco Zaccagnino[2], and Rosalba Zizza[2]

[1] Dip. di Informatica, Sistemistica e Comunicazione, University of Milano-Bicocca, viale Sarca 336, 20126 Milan, Italy
paola.bonizzoni@unimib.it, a.petescia@campus.unimib.it, yuri.pirola@unimib.it, raffaella.rizzi@unimib.it

[2] Dip. di Informatica, University of Salerno, via Giovanni Paolo II 132, 84084 Fisciano, Italy
{rzaccagnino, rzizza}@unisa.it

Abstract. Detecting common regions and overlaps between DNA sequences is crucial in many Bioinformatics tasks. One of them is genome assembly based on the use of the overlap graph which is constructed by detecting the overlap between genomic reads. When dealing with long reads this task is further complicated by the length of the reads and the high sequencing error rate. This paper proposes a novel alignment-free method for detecting the overlaps in a set of long reads which exploits a signature (called *fingerprint*) of reads built from a factorization of the read based on the notion of Lyndon words. The method has been implemented in the tool `KFinger` and tested over a simulated and a real PacBio HiFi dataset of genomic reads; its results have been compared with the well-known aligner `Minimap2`. `KFinger` is available at https://github.com/AlgoLab/kfinger.

Keywords: Lyndon word · Factorization · Fingerprint · Overlap graph · Long reads.

1 Introduction

Lyndon word is a concept of combinatorics on words and a well-known notion in Bioinformatics [1, 2], where it has been used to find short motifs [3] and more recently in the notion of the extended BWTs [4]. Most notably, a recent work suggests that Lyndon factorizations can be used to detect overlaps between reads [5], which is the fundamental task to build the overlap graph in genome assembly. Note that a factorization (as notion of combinatorics on words) expresses

* This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 872539.
The final authenticated version is available online at https://doi.org/10.1007/978-3-031-07802-6_37.

a string as a concatenation of factors and factors in a Lyndon factorization are Lyndon words. Lyndon factorization [1, 6] is one of the most well-known factorizations and has two main properties: (i) it is unique for a given string and (ii) can be computed in linear time. Moreover, it satisfies the following crucial property, which is the foundation of our proposed method: two strings sharing a common overlap also share a set of consecutive common factors in their factorizations [5].

Detecting the overlap between sequences is the fundamental step in de-novo assembly based on the Overlap-Layout-Consensus (OLC) strategy [7], which is the main approach used for assembling long reads [8,9]. Since unfortunately such reads are long and error-prone, detecting overlaps is often a bottleneck from a computational point of view, mainly when a pairwise comparison is adopted, due to the fact that long reads have high sequencing errors and contain repetitive regions. Several methods for discovering overlaps in long reads, which are based on a representation of the input reads, are present in literature, achieving good performance in terms of computation time and accuracy. For example, [10] proposes an algorithm combining minimizers and MinHash algorithm [11] for mapping long reads to a reference database; sourmash [12] and MHAP [8] use MinHash algorithm (MHAP relies on k-mers); sourmash estimates sequence similarity between very large data sets whereas MHAP is a tool for discovering overlaps between long reads and is used by Canu assembler [13]. Minimap2 [14] is an aligner of DNA or mRNA long reads against a large reference database and uses minimizers.

We propose an alignment-free approach for discovering the overlaps in a set of noisy long reads, exploiting a compact representation (or signature) given by the sequence of lengths of the Lyndon factors (instead of the factors themselves) in Lyndon factorizations. The sequence of factor lengths, called *fingerprint*, has been first introduced in [15] as a mean to discover common regions between reads and applied for classifying RNA-Seq reads by origin gene. Read fingerprints provide a compact representation of the reads and unexpectedly they are effective in preserving sequence similarities, thus being extremely useful in an alignment-free approach for discovering similarities. The main idea is that a factorization of a read is computed while reading the reads and the factorization splits the reads based on their content in terms of Lyndon-words: we keep the sequence of the distances between consecutive splitting positions (that is, the sequence of the factor lengths) to use as read fingerprint (read signature). The k-mers of a fingerprint (called *k-fingers*) are the sub-pieces able to to capture the similarity regions between the reads in a more flexible way with respect to the k-mers of a sequence: indeed the length k of a k-mer is fixed. Furthermore, fingerprints (numerical sequences) are shorter than the represented nucleotide sequences and we expect that they are also resilient to errors occurring in long reads and common k-fingers can be discovered. In the paper we show that k-fingers provide anchors for computing common regions between reads of an input set S and present an algorithm performing factorization of the reads in S and (next) a linear scanning of the read signatures (or fingerprints); by hashing the k-fingers,

the common regions, shared by the currently processed read s and all the reads previously considered, are computed in $O(LN)$, where L is the read length and N is the maximum number of occurrences of a unique (occurring once) k-finger of s in the reads considered by the previous iterations. At the end of the iterations, the algorithm has computed all the common regions between the input reads. Observe that comparing reads in a reference-free approach often requires a pairwise comparison and is computationally demanding (refer for example to the problem of the identification of the relationships between metagenomic reads [16]). We have implemented our method in the Python prototype `KFinger` taking as input a set of reads and producing as output the pairs of reads in overlap. We have tested it over an error-free dataset of long reads simulated from a 2M-long region of the human chromosome 21 by using `DeepSimulator` [17] and a real PacBio HiFi set. We have compared the results from `KFinger` and `Minimap2` [14].

Overall, `Minimap2` produces more overlapping pairs than `KFinger` and the percentage of overlaps with high error rate (error rate over 3.0%) is higher for `Minimap2` than `KFinger`. Observe that pair of reads, that share a short overlap, are expected to be missed by our method, but, on the other end, with the purpose of reconstructing an assembly, these pairs of reads may be discarded. The obtained results suggest that `KFinger` is less sensitive than `Minimap2` in the face of a quite high specificity. To test this hypothesis, we also compared the results (by `KFinger` and `Minimap2`) with the overlaps obtained by mapping the input reads to the reference genome.

2 Preliminaries

Let $s = c_1 \cdots c_p$ be a string over a finite alphabet Σ. The *length* of s (that is, the number p of its characters) will be denoted by $|s|$. A *prefix* of s is a string composed of its first i characters (that is, $c_1 \cdots c_i$). Similarly, a *suffix* is a string composed of the last i characters of s (that is, $c_{n-i+1} \cdots c_n$).

A prefix (or suffix) is *proper* if it does not cover the whole string s. In the following, notation $s < s'$ (resp. $s \leq s'$) will specify that string s is lexicographically smaller than s' (resp. or $s = s'$). Furthermore, $s \ll s'$ will specify that $s < s'$ and additionally s is not a proper prefix of s'.

Now, we introduce the two main ingredients for capturing common regions between two strings (or reads): the definitions of *factorization* and *fingerprint*. Precisely, a *factorization* of a string s is a sequence $F(s) = \langle f_1, f_2, \ldots, f_n \rangle$ of factors (strings over Σ), such that $s = f_1 f_2 \cdots f_n$ and the *fingerprint*, with respect to $F(s)$, is the sequence $\mathcal{L}(s) = \langle |f_1|, |f_2|, \ldots, |f_n| \rangle$ of the factor lengths.

Given a fingerprint $\mathcal{L}(s) = \langle l_1, l_2, \ldots, l_n \rangle$, a *k-finger* is a k-mer of $\mathcal{L}(s)$, that is, any substring $\langle l_i, l_{i+1}, \ldots, l_{i+k-1} \rangle$ composed of k consecutive elements of $\mathcal{L}(s)$. The sum $l_i + l_{i+1} + \cdots + l_{i+k-1}$ will be referred as *supporting length* of the k-finger. Moreover, the index i and the sum $l_1 + l_2 + \cdots + l_{i-1}$ of the upstream elements (lengths) of the fingerprint will be referred as *index offset* and *length offset* of the k-finger with respect to the fingerprint.

The substring $f_i f_{i+1} \cdots f_{i+k-1}$ will be the *supporting string* of the k-finger.

Example 1. Let $F(s) = \langle aaaaa, cccc, \mathbf{aaaaaa, ccccc, ttt}, a\rangle$ be the factorization of s and let $\mathcal{L}(s) = \langle 5, 4, \mathbf{6, 5, 3}, 2, 1\ \rangle$ be the the fingerprint. The three bold consecutive integers $\langle \mathbf{6, 5, 3}\rangle$ are a 3-finger, whose supporting length is 14 and supporting string is the concatenation of the three bold factors of the factorization. The *index offset* of the 3-finger is 3, since its first element is the third in the whole fingerprint, and the *length offset* is 9, which the sum of the upstream elements 5 and 4. The *length offset* gives the offset of the supporting string in s.

In order to obtain read fingerprints, in this work we will exploit special kinds of factorizations, named *Lyndon based factorizations* in [15] since they are defined starting from the well known Lyndon factorization of a string s [1]. We firstly recall that each string s can be uniquely factorized into *Lyndon words* [1], where a Lyndon word is a word which is strictly smaller than any of its non empty proper suffixes. For example, it is easy to see that *accgctct* is a Lyndon word, whereas *cac* is not a Lyndon word. Formally, given a string s, its Lyndon factorization is denoted by $\mathsf{CFL}(s) = \langle f_1, f_2, \ldots, f_n\rangle$, where $f_1 \geq f_2 \geq \cdots \geq f_n$ and each f_i is a Lyndon word. For example, given $s_1 = gcatcaccgctctacagaac$, we have that $\mathsf{CFL}(s_1) = \langle g, c, atc, accgctct, acag, aac\rangle$. In [18], the *Canonical Inverse Lyndon factorization* $\mathsf{ICFL}(s) = \langle f_1, f_2, \ldots, f_n\rangle$ is a factorization of s such that $f_1 \ll f_2 \ll \cdots \ll f_n$ and each f_i is an *inverse Lyndon word* [18], that is, each non empty proper suffix of f_i is strictly smaller than f_i. For example, *cac*, *tcaccgc* are inverse Lyndon words. Let us consider again $s_1 = gcatcaccgctctacagaac$. We have that $\mathsf{ICFL}(s_1) = \langle gca, tcaccgc, tctacagaac\rangle$. Such factorizations are unique and can be computed in linear time and constant space [18].

A property of $\mathsf{CFL}(s) = \langle f_1, f_2, \ldots, f_n\rangle$, which is crucial in our framework, is the following *Conservation Property* [19]. Suppose that $\mathsf{CFL}(s) = \langle f_1, f_2, \ldots, f_n\rangle$ and let $z = f'_l f_{l+1} \cdots f_t f'_{t+1}$ be a non simple factor w.r.t. $\mathsf{CFL}(s)$ (i.e., it properly contains at least one factor), for some indexes l, t with $1 \leq l < n$, $1 < t < n$, and $f_l = f''_l f'_l$, $f_{t+1} = f'_{t+1} f''_{t+1}$. A main consequence of the conservation property proved in [18] is that given two strings s and s' sharing a common overlap z, there exist factors that are in common between $\mathsf{CFL}(w)$ and $\mathsf{CFL}(w')$. Thus s and s' they will have fingerprints sharing k-fingers for a suitable size k. For example, consider again $s_1 = gcatcaccgctctacagaac$ and let $s_2 = ccaccgctctacagaagcatc$. We know that $\mathsf{CFL}(s_1) = \langle g, c, atc, accgctct, acag, aac\rangle$ and we have that $\mathsf{CFL}(s_2) = \langle c, c, accgctct, acag, aagcatc\rangle$. Hence, we have $\mathcal{L}(s_1) = \langle 1, 1, 3, 8, 4, 3\rangle$ and $\mathcal{L}(s_2) = \langle 1, 1, 8, 4, 7\rangle$. The two common consecutive elements $\langle 8, 4\rangle$ are related to the same factors in the strings (8 is realated to *accgctct* and 4 is related to *acag*) and capture the common substring *accgctctacag* given by their concatenation.

Our method exploits the previous result and is based on the following assumption: a k-finger occurring in different read fingerprints has the same supporting string. This assumption is fundamental in order to capture common regions between reads by using fingerprints and k-fingers while ignoring the string characters. We define CFL_ICFL the factorization obtained by applying first the Standard Lyndon Factorization CFL, and then the Canonical Inverse Lyndon factorization ICFL to factors (of CFL) longer than a given threshold. In other words, given

$\mathsf{CFL}(s) = \langle f_1, f_2, \ldots, f_n \rangle$, we obtain $\mathsf{CFL\_ICFL}(s)$ by replacing with $\mathsf{ICFL}(f_i)$ each f_i longer than the threshold.

Observe that CFL_ICFL has the main advantage of producing many factors, thus enriching the set of k-fingers to use for detecting the common regions between reads. In [15], in order to deal with the double-stranded nature of sequencing reads it is proposed a factorization algorithm $F^d(s) = \langle f_1, f_2, \ldots, f_n \rangle$ such that $F^d(\overline{s})$ is equal to $\langle \overline{f}_n, \overline{f}_{n-1}, \ldots, \overline{f}_1 \rangle$, where $\overline{f}_i$ is the reverse and complement of f_i. Recall that the *reverse and complement* of a string s over the DNA alphabet $\{A, C, G, T\}$ is the string $\overline{s}$, such that its i-th character is the complement of the $(|s| - i + 1)$-th character of s, where the *complement* is the operation transforming the DNA symbol A into the DNA symbol T (and vice versa) and the DNA symbol C into the DNA symbol G (and vice versa). This double-stranded factorization relies on a basic algorithm F such as CFL, ICFL or CFL_ICFL, and is obtained by combining $F(s)$ with $F(\overline{s})$, with the result of reducing the length of the factorization factors [15].

Observe that fingerprint of s will be equal to the reverse of fingerprint of $\overline{s}$ and, as a consequence, the same genomic region on the two opposite strands will be supporting two k-fingers, which are one the reverse of the other.

3 Detecting reads in overlap

In our framework, we consider in overlap two reads s and s' between which, one of the following relations occur: (i) a proper suffix of s has a match with a proper prefix of s' (or vice versa), (ii) s has a match with a substring of s' (or vice versa). In absence of sequencing errors, the suffix of s will be equal to the prefix of s' (or vice versa) in case (i) and s will be equal to the substring of s' (or vice versa) in case (ii). Clearly, when sequencing errors are present, the equality relation must be transformed into a similarity relation . Observe that the above relation (i) holds for two reads sequenced from the same genomic strand. When the reads come from opposite strands, then relation (i) must be turned into the following one: a proper suffix (resp. prefix) of s has a match with a proper suffix (resp. prefix) of s' (or vice versa). Obviously, the matching in both cases occurs except for a reverse and complement operation of one of the two involved read substrings. Our aim is to use fingerprints and k-fingers obtained from Lyndon-based factorizations for capturing common regions between reads in an input set and inferring pairs of reads in overlap. Given an input set of reads, our method duplicates each input read. In other words, we expand the input set by adding the reverse and complement version of each input read. Then, it computes for each read (of the expanded set) a Lyndon-factorization from which to obtain the fingerprint (read signature) and extract the k-fingers. Next, it exploits the obtained k-fingers to detect common regions between reads and infer the pairs of reads in overlap (or overlapping pairs) in the expanded input set. Observe that, following the duplication approach to handle the double-stranded nature of reads, we only have to deal with suffix-prefix overlaps as if the reads originated from the same strand. Next, a post-processing step obtains the overlapping pairs of the

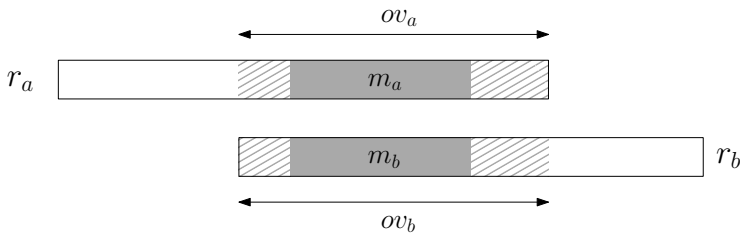

Fig. 1. Suffix-prefix overlap between reads r_a and r_b having common region (m_a, m_b). A suffix of r_a has a match with a prefix of r_b.

original input set and (if needed) converts suffix-prefix overlaps into suffix-suffix (or prefix-prefix) overlaps between reads from opposite strands.

3.1 The method

Let $S = \{s_1, s_2, \ldots, s_r\}$ be the set of the input reads (strings over the DNA alphabet) and let $\overline{s}_i$ be the reverse and complement of s_i. The set S is first expanded into the set $S_e = \{s_1, \overline{s}_1, s_2, \overline{s}_2, \ldots, s_r, \overline{s}_r\}$. Then (first step), each read in S_e is split into segments of a given length $\mathcal{X}$ (observe that the last segment may be smaller) and each segment is factorized by using a factorization algorithm (among the ones described in the previous section). The fingerprint of a read will be the concatenation of the fingerprints of its segments. The read segmentation has the advantage of producing richer fingerprints in terms of number of elements and therefore in terms of k-fingers to use to capture similarities. Next (second step), the read fingerprints are exploited to obtain the pairs of reads (of the expanded set S_e) sharing a common region. Observe that we are not interested in overlapping pairs $(s_i, \overline{s}_i)$ composed of a read and its reverse and complement. This step considers pairs (r_a, r_b) such that r_a is s_i or $\overline{s}_i$ and r_b is in $\{s_{i+1}, \overline{s}_{i+1}, \ldots, s_r, \overline{s}_r\}$ (the vice versa is indeed redundant). This step basically finds two common unique k-fingers (occurring uniquely in the two reads) to use as anchors of the common region between r_a and r_b. For each computed common region (third step), the suffix-prefix overlap is obtained by extending the common region to the left endpoint of a read and to the right endpoint of the other read (as depicted in Figure 1. When the common region does not cover a certain percentage P of the putative overlap, then the pair (r_a, r_b) is not an overlapping pair and will not be produced as output.

Finally (fourth step), after computing all the suffix-prefix overlaps of the expanded set S_e, a post-processing step computes the overlapping pairs of the original input set S. Precisely, let s_i, s_j and $\overline{s}_i$, $\overline{s}_j$ be two input reads and their reverse and complement versions. Assuming $i < j$, then the overlapping pairs (s_i, s_j), $(s_i.\overline{s}_j)$, $(\overline{s}_i, s_j)$ and $(\overline{s}_i, \overline{s}_j)$ may be coexist in the output of the third step. Hence, a trivial strategy is applied to only retain just one among those pairs, which consists in selecting the first pair produced by the algorithm. Observe that sophisticated strategies have been tested (using some criteria based on the read

strand) but we did not obtain a significant improvement in the results. Observe that when the selected pair is $(s_i.\overline{s}_j)$ or $(\overline{s}_i, s_j)$ (that is, it involves reads from opposite strands), then the suffix-prefix overlap is converted into a suffix-suffix or prefix-prefix overlap. When the selected pair is $(\overline{s}_i, \overline{s}_j)$, the suffix-prefix overlap is reported onto the original reads s_i and s_j.

The following paragraphs are devoted to detail the second step which is the core of our method and works in two sub-steps: first, the *candidate pairs* are computed (see Algorithm 1) and then the common regions are obtained. Basically, Algorithm 1 performs a linear scanning of the reads of S_e and, for each read fingerprint, the k-fingers are considered from the leftmost to the rightmost. The goal is to compute a hash table C storing the pairs (r_a, r_b) sharing at least U unique k-fingers (that is, occurring only once in both reads), which are referred as *candidate pairs*. The leftmost (unique) k-finger shared by r_a and r_b is stored in C for each candidate pair (r_a, r_b) together with its *length offsets* and *index offsets* in the fingerprints of two reads. The returned hash table C gives for a key (r_a, r_b) (candidate pair) the tuple $(f_l, \omega_a^l, i_a^l, \omega_b^l, i_b^l)$, where f_l is the common leftmost k-finger, ω_a^l and ω_b^l are the length offsets for r_a and r_b (respectively) and i_a^l and i_b^l are the index offsets for r_a and r_b (see Example 1). The algorithm uses a support hash table H storing the k-fingers and their localization in the reads (length offset and index offset): for each k-finger f, the value $H(f)$ is a list of tuples (r, ω, i), where each tuple gives the localization of f in the fingerprint of a read r. For each considered read r_b (see the main `foreach` cycle at line 3) and for each k-finger f, its localization in r_b is stored in the hash table H (see `foreach` cycle at line 5). Then, H is updated such that it contains only the localizations of the unique k-fingers of r_b (see `foreach` cycle at line 9) and at the same time such unique k-fingers are stored in the list *unique_list*. The `if` condition at line 10 checks whether the k-finger f is unique in r_b. In fact, if f is not unique, then the $n > 1$ trailing tuples of list $H(f)$ will be related to r_b. At each iteration of the main `foreach` cycle, the support hash table H contains, for each read already processed before r_b, only the localizations of its unique k-fingers. The last `foreach` cycle at line 15 considers each unique k-finger of r_b and finds its localizations in the other reads (processed before r_b) in order to compute all the candidate pairs involving r_b as second read. Observe that the k-fingers are always considered from left to right in the read fingerprints and the two `foreach` cycles at lines 9 and 15 guarantee that the k-finger f, stored in C for a candidate pair (r_a, r_b), is the leftmost unique k-finger shared by the two reads. Algorithm 1 performs a linear scanning of the read fingerprints and the three `foreach` cycles at lines 5, 9 and 15 perform a linear scanning of the read k-finger whose number is asymptotically equal to the read length. Finally, observe that the `foreach` cycle at line 16 only checks the tuples in the list $H(f)$ (of the support hash table H) whose size is the number of reads (among the ones already processed) containing a unique occurrence of the k-finger f. Even though it is not specified by the algorithm, only k-fingers whose *supporting length*, i.e. the sum of the lengths in the k-finger, is at least a given threshold τ are considered. The parameter τ is the threshold we use to consider a k-finger reliable and avoid

Algorithm 1: Compute the candidate pairs

Input : Fingerprints of the reads of the expanded set S_e
Output: C, hash table of the candidate pairs

$H \leftarrow$ empty hash table;
$C \leftarrow$ empty hash table;
foreach *fingerprint* $\mathcal{L}$ **do**
  $r_b \leftarrow$ read whose fingerprint is $\mathcal{L}$;
  foreach *k-finger* $f \in \mathcal{L}$ **do** // From the leftmost to the rightmost
    $(\omega, i) \leftarrow$ length offset and index offset of f;
    Add (r_b, ω, i) to the list $H(f)$;
  unique_list $\leftarrow$ empty list;
  foreach *k-finger* $f \in \mathcal{L}$ **do**
    if *the last* $n > 1$ *tuples of* $H(f)$ *are related to* r_b **then**
      Remove from $H(f)$ the last n tuples;
    else // f is unique in r_b
      Append f to *unique_list*;
  already_processed $\leftarrow$ empty set;
  foreach $f \in$ *unique_list* **do**
    foreach $(r_a, \omega_a^l, i_a^l) \in H(f)$ **do**
      if $r_a \neq r_b$ *and* $r_a \notin$ *already_processed and* $(r_a, r_b) \notin C$ **then**
        if r_a *and* r_b *share at least* U *unique k-fingers* **then**
          $(\omega_b^l, i_b^l) \leftarrow$ length offset and index offset of f in $\mathcal{L}$;
          $C(r_a, r_b) \leftarrow (f, \omega_a^l, i_a^l, \omega_b^l, i_b^l)$;
        else
          Add r_a to *already_processed*;
return C

collisions (that is, the same k-finger which is supported by different strings in different reads).

For each candidate pair (r_a, r_b) in the hash table C, the algorithm uses the tuple $(f_l, \omega_a^l, i_a^l, \omega_b^l, i_b^l)$ returned by C to localize the two longest subsequences (consecutive elements) of fingerprints $\mathcal{L}(r_a)$ and $\mathcal{L}(r_b)$ of r_b which satisfy the following three conditions: (1) both subsequences have k-finger f_l as prefix, (2) they share at least the last k' elements (where k' is an input parameter) and the k'-finger corresponding to such elements uniquely occurs in the two reads and has a minimum supporting length (to avoid collisions), and (3) the sum of the elements (integer values) of the first subsequence differs from the sum of the elements of the second subsequence by an upper threshold, we call *length tolerance*. The algorithm further extends as much as possible on the right the two subsequences while maintaining the equality of the corresponding elements.

Example 2. Let $\mathcal{L}(r_a) = \langle 5, 4, 3, \mathbf{10, 6, 5, 3, 2, 7, 3}, 4\rangle$ be the fingerprint of r_a and let $\mathcal{L}(r_b) = \langle 2, 2, \mathbf{10, 6, 3, 2, 2, 2, 7, 3}, 3, 9\rangle$ be the fingeprints of r_b. Assuming $k = 2$, $k' = 2$, a *length tolerance* set to 3 and a minimum supporting length set to 10 for k-fingers, then the bold subsequences satisfy the above conditions. Indeed, both ones start with the 2-finger $\langle 10, 6\rangle$ which is the leftmost

common k-finger (occurring just once in the reads) having a supporting length at least 10. Moreover, they share the last k'-finger (the last k' elements) $\langle 7, 3\rangle$ having a supporting length at least 10 (assuming that 10 is also the minimum supporting length for the k'-finger terminating such subsequences. Finally, the sum of the bold subsequence of r_a is equal to 36, while the sum of the bold subsequence of r_b is equal to 35 and their difference satisfies the assumed length tolerance. Hence, the common region between r_a and r_b (computed by our method) will be composed of the 36-long substring starting at position $5 + 4 + 3 + 1 = 13$ of r_a and the 35-long substring starting at position $2 + 2 + 1 = 5$ of r_b. At this point, the common region between the reads is obtained by finding the two read substrings, referred in the following as *common region*, supporting the two computed fingerprint subsequences. The *length tolerance* admitted in condition (3) takes into account possible sequencing errors of the reads and is the maximum difference between the length of the two detected common read substrings. Observe that the two fingerprint subsequences may share only a prefix (the leftmost k-finger f_l) and a suffix and the equality of the corresponding integers may be interrupted because of sequencing errors or read segmentation (see the first step). Our method also allows to perform a *re-factorization* of one of the two reads r_a and r_b before computing the core common region, motivated by the fact that the read segmentation (see step one) may lead to a misalignment in the segment fingerprints, thus inducing to lose common factors in the overlapping regions between two reads.

During re-factorization, the read (between r_a and r_b), where the common leftmost k-finger f_l has the smallest length offset, is selected; the suffix w to re-factorize is computed as described by Figure 2. Note that w is aligned with a factorization segment of the other read and therefore the fingerprint of w will be compared with the suffix of the fingerprint of the other read starting from such segment. The common region between r_a and r_b will be computed, as described before, starting from the a common leftmost k-finger shared by the two new fingerprints. In case of re-factorization, the common region between the two reads will be the longest between the ones computed before and after re-factorization.

4 Experimental Results

The method has been implemented in the Python prototype `KFinger` and it is available at https://github.com/AlgoLab/kfinger along with all the scripts needed to replicate the experiments. The tests have been performed on an Ubuntu 20.04 laptop with a single Intel® Core™ i5-8250U CPU and 16GB of RAM over the following datasets: (1) a dataset of 10K error-free long reads simulated from the region of the human chromosome 21 between positions 32 000 000 and 34 000 000 (2 000 000bp), by using `DeepSimulator` [17] and (2) a dataset of 6141 real PacBio HIFI reads extracted from PacBio Sequel II HiFi sequencing of sample HG00731 of a Puerto Rican Trio. Precisely, the reads were mapped against the human genome GRCh38 (GCA_000001405.15, no ALT contigs) using `Minimap2` (version 2.17) with preset asm20 (as suggested in its documentation for aligning PacBio

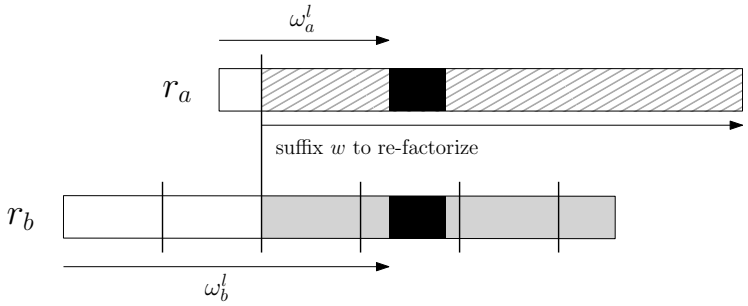

Fig. 2. Re-factorization scheme. The two reads r_a and r_b are depicted as horizontal bars aligned according to the the common leftmost k-finger f_l whose supporting strings are depicted as black boxes. ω_a^l and ω_b^l are the length offsets of f_l and the vertical bars on r_b (which has the largest length offset for f_l) are the edges between consecutive factorization segments (only edges in the portion aligned with r_a are shown). The leftmost edge falling in r_a determines the starting point of the suffix w of r_a to re-factorize (highlighted in tiled grey), whose fingerprint will be compared with the fingerprint of r_b corresponding to the portion highlighted in solid grey.

HiFi/CCS genomic reads). Only primary alignments were retained. We then extracted in FASTA format reads overlapping region 32M–34M of chromosome 21. The final dataset was composed by 6141 reads with average length 11124bp (min 2349bp, max 23263bp, median 10417bp) for a total of 68316199 bases. The error rate inferred from the alignment was 6.22×10^{-3}. Original sequence files are available at http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data_collections/HGSVC2/working/20190925_PUR_PacBio_HiFi/
(run IDs: r54329U_190528 – r54329U_190906).

The first dataset will be referred as `error-free-ds` while the second one as `hifi-ds`. Recall that each input read to `KFinger` is accompanied by its reverse and complement, so that the size of the two datasets is 20000 for `error-free-ds` and 12282 for `hifi-ds`. We have used a double-stranded factorization algorithm built over the basic CFL_ICFL factorization algorithm with threshold parameter 30 (that is, factors of CFL factorization longer that 30 are submitted to ICFL factorization), by splitting each read into segments of length $\mathcal{X} = 300$bp in order to limit the factor lengths. The common regions between reads were computed for both datasets before and after re-factorization; then, the overlaps from common regions covering at least a percentage $P = 80\%$ (coverage percentage of the putative overlap), were obtained. For finding the candidate pairs, we used a k-finger size set to 7 ($k = 7$) and a minimum supporting length set to 40; 6 is the minimum number of unique shared k-fingers required for a candidate pair. Moreover, before re-factorization, we required $k' = 3$ and a length tolerance set to 0 for `error-free-ds` and $k' = 2$ and length tolerance set to 15 for `hifi-ds`. After re-factorization, we used $k = 5$ and a minimum supporting length set to 10. We maintained the above values for parameter k' for the two datasets and the length tolerance set to 0 for `error-free-ds`, while setting to 20 the length tolerance for

hifi-ds. We have compared, in terms of accuracy, KFinger with Minimap2 [14] by retaining only the common regions produced by Minimap2 having a minimum coverage percentage $P = 80\%$ with respect to the putative overlap and computing the overlaps on such common regions. Observe that records involving the same read have been discarded both for KFinger and Minimap2. For each common region and each overlap obtained with Minimap2 and KFinger, we computed an error rate as the ratio of the edit distance, between the two read substrings involved in a common region or an overlap, and the smaller substring length. Tables 1 and 2 report the results for datasets error-free-ds and hifi-ds, respectively. Both tables report the results on common regions and overlaps produced by KFinger before (rows **K**) and after re-factorization (rows **KR**) and on common regions and overlaps obtained from Minimap2 (rows **M**). In the following, we refer to a common region or an overlap with the generic term *record*. The first three columns "#0", "#$\leq$ 3.0" and "#$>$ 3.0" give the number of records having an error rate equal to 0, greater than 0 but at most 3.0% and over 3.0%, respectively. The last three columns MinL, MaxL and AvgL report the minimum, maximum and average length of the read substrings involved in the record. Overall, Minimap2 finds more records than KFinger. Over error-free-ds Minimap2 outputs a total of 1286932 common regions, 179757 out of them are alternative overlaps between reads, against the 533584 produced by KFinger. Observe that a given pair of reads may be involved in more than one record; only Minimap2 produces alternatives, whereas KFinger gives (by choice) just one common region/overlap for two given reads. Them, over hifi-ds, Minimap2 finds a total of 530529 common regions (108655 out of them are alternatives) against the 211230 produced by KFinger. Moreover, Minimap2 finds a total of 502377 overlaps (4455 out of them are alternatives) over error-free-ds and a total of 132160 overlaps (461 out of them are alternatives) over hifi-ds. KFinger produces 433947 overlaps over error-free-ds and 147029 overlaps over hifi-ds before re-factorization, whereas it produces 465121 overlaps over error-free-ds and 173057 overlaps over hifi-ds after re-factorization.

Column "#$>$ 3.0" reports in parentheses the percentage of records (having an error rate over 3.0%) with respect to the total number of obtained records. We consider this value as a proxy of the false positive rate of the prediction. Observe that this percentage is rather low for KFinger both for common regions and overlaps, whereas for Minimap2 it is low only for the overlaps since the parameter $P = 80\%$ contributes to filter out the common regions produced by Minimap2 not leading to a good read overlap. Moreover, the re-factorization has mainly determined an improvement in terms of detected overlaps, since for dataset error-free-ds 31079 extra overlaps (with an error rate ≤ 3.0) were detected after the re-factorization with respect to the experiment before re-factorization. Similarly, the re-factorization has produced 25486 extra overlaps for dataset hifi-ds. These results suggest that KFinger does not compete with Minimap2 in terms of sensitivity but it is likely to be more specific in terms of common regions. Indeed, Minimap2 is more tolerant with respect to the sequencing errors and therefore finds more common regions than KFinger. On the other hand,

Table 1. Experimental results for `error-free-ds`. The rows tagged as **K** and **KR** refer to the common regions/overlaps produced by `KFinger` before and after re-factorization, whereas the row tagged as **M** refers to common regions/overlaps obtained from `Minimap2`.

Common regions

	#0	#≤ 3.0	#> 3.0	**MinL**	**MaxL**	**AvgL**
K	473053	7643	52888 (10%)	40	37383	4339
KR	474107	7103	52374 (10%)	39	37414	4550
M	498479	29577	758876(59%)	100	37441	2246

Overlaps ($P = 80\%$)

	#0	#≤ 3.0	#> 3.0	**MinL**	**MaxL**	**AvgL**
K	433884	8	55 (0.1%)	95	37441	5622
KR	464958	13	150 (0.1%)	95	37441	5364
M	496289	2142	3946(1%)	100	37441	5055

`KFinger` gives fewer common regions and seems to be more precise. To test this hypothesis, and, in particular, to evaluate sensitivity, we compared the predicted common regions with the overlaps computed by mapping reads to the reference genome. We mapped the two datasets to region 32M–34M of human chromosome 21 using `Minimap2` and we kept only reads aligning for at least 95% of their length. From these alignments we devised the set of overlaps such that the length of the overlap was at least 80% of the length of the genomic region spanned by the two reads. We define this set as the set of "alignment-based" overlaps. Please notice that we do not expect that the set of alignment-based overlaps coincides with the set of predicted overlaps since (i) some overlaps were discarded because of their length and since (ii) there exists common regions between reads that do not actually overlap on the genomic. For each alignment-based overlap, we checked if there exists a predicted common region that intersects the overlap for at least 50% of their span. If it exists, we considered the alignment-based overlap as found. The dataset `error-free-ds` contains 9273 alignment-based overlaps. As expected, `Minimap2` found all of them, while `KFinger` missed 5 of them before re-factorization and 3 of them after re-factorization. The dataset `hifi-ds` contains 16207 alignment-based overlaps. `Minimap2` was not able to find 2 of them, while `KFinger` missed 1743 of them before re-factorization and 753 of them after re-factorization. These results support the hypothesis that `Minimap2` is more sensitive and more tolerant than `KFinger`, but, on the other hand, it is also less specific, since `Minimap2` reports twice as much common regions as `KFinger`. In terms of time efficiency, we measured the whole time for computing the candidate pairs and the common regions. These two steps are indeed the intensive part of the method. Moreover, the time is given before re-factorization, since the current implementation of the read factorization is not optimal. On a single thread, `KFinger` took 12 minutes and 5 seconds for

Table 2. Experimental results for `hifi-ds`. The rows tagged as **K** and **KR** refer to the common regions/overlaps produced by `KFinger` before and after re-factorization, whereas the row tagged as **M** refers to common regions/overlaps obtained from `Minimap2`.

Common regions

	#0	#≤ 3.0	#> 3.0	**MinL**	**MaxL**	**AvgL**
K	10309	184461	16460 (8%)	40	17853	4392
KR	6976	187396	16858 (8%)	39	18063	4933
M	9449	217870	303210(57%)	100	18811	2531

Overlaps ($P = 80\%$)

	#0	#≤ 3.0	#> 3.0	**MinL**	**MaxL**	**AvgL**
K	2583	143916	530 (0.3%)	97	18169	6036
KR	3275	168710	1072 (0.1%)	97	18169	5880
M	2646	109115	20399(15%)	103	19664	6623

dataset `error-free-ds` and 4 minutes and 2 seconds for dataset `hifi-ds`. Despite being highly optimized, `Minimap2` took 4 minutes and 42 seconds for dataset `error-free-ds` and 2 minutes and 16 seconds for dataset `hifi-ds`.

5 Conclusions and Future Developments

We have proposed a method for detecting overlaps in a set of long reads by using a compact numerical representation (fingerprint) based on Lyndon factorization. The method has been implemented in the Python prototype `KFinger` which has been tested over a set of error-free simulated reads and a PacBio HIFI dataset. The experimental results encourage to think that `KFinger` may be a suitable and specific method for finding shared regions between pairs of reads, taking advantage of the compact numeric representation of the reads. In the immediate we plan to improve `KFinger` in terms of time efficiency by improving the implementation of (1) the factorization algorithms used for producing the input fingerprints and (2) of the steps two and three producing the common regions, improvement needed in terms of a more efficient programming language such as C++ and the use of more efficient data structures. In terms of accuracy we plan to investigate the impact of the different factorization algorithms in order to face the typical issues related to long reads: sequencing errors and repetitive regions.

References

1. Lyndon, R. C.: On Burnside's problem. Transactions of the American Mathematical Society **77**(2), 202–215 (1954)
2. Berstel, J., Perrin, D.: The origins of combinatorics on words. European Journal of Combinatorics **28**(3), 996–1022 (2007)

3. Delgrange, O., Rivals, E.: Star: an algorithm to search for tandem approximate repeats. Bioinformatics **20**(16), 2812–2820 (2004)
4. Mantaci, S., Restivo, A., Rosone, G., Sciortino, M.: An extension of the Burrows–Wheeler transform. Theoretical Computer Science **387**(3), 298–312 (2007)
5. Bonizzoni, P., De Felice, C., Zaccagnino, R., Zizza, R.: Lyndon words versus inverse Lyndon words: Queries on suffixes and bordered words. In: LATA 2020, LNCS, vol. 12038, pp. 385–396. Springer (2020). https://doi.org/10.1007/978-3-030-40608-0_27
6. Chen, K. T., Fox, R. H., Lyndon, R. C.: Free Differential Calculus, IV. the quotient groups of the lower central series. Annals of Mathematics **68**(1), 81–95 (1958)
7. Pevzner, P. A., Tang, H., Waterman, M. S.: An Eulerian path approach to DNA fragment assembly. In: Proceedings of the National Academy of Sciences, pp. 9748–9753. National Academy of Sciences, 98(17) (2001)
8. Berlin, K., Koren, S., Chin, C.-S., Drake, J. P., Landolin, J. M., Phillippy, A. M.: Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. Nature biotechnology **33**(6), 623–630 (2015)
9. Loman, N. J., Quick, J., Simpson, J. T.; A complete bacterial genome assembled de novo using only nanopore sequencing data. Nature Methods **12**(8), 733–735 (2015)
10. Jain, C., Dilthey, A., Koren, S., Aluru, S., Phillippy, A. M.: A fast approximate algorithm for mapping long reads to large reference databases. Journal of Computational Biology **25**(7), 766–779 (2018)
11. Broder, A.: On the resemblance and containment of documents. In: Proceedings. Compression and Complexity of SEQUENCES, pp. 21–29. IEEE Comput. Soc (1997)
12. Pierce, N. T., Irber, L., Reiter, T., Brooks, P., Brown, C. T.: Large-scale sequence comparisons with sourmash. F1000Research **8**, 1006 (2019)
13. Koren, S., Walenz, B. P., Berlin, K., Miller, J. R., Bergman, N. H., Phillippy, A. M.: Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. Genome Research **27**(5), 722–736 (2017)
14. Li, H.: Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics **34**(18), 3094–3100 (2018)
15. Bonizzoni, P., De Felice, C., Petescia, A., Pirola, Y., Rizzi, R., Stoye, J., Zaccagnino, R., Zizza, R.: Can we replace reads by numeric signatures? Lyndon fingerprints as representations of sequencing reads for machine learning. In: AlCoB 2021. LNCS, vol. 12715, pp. 16–28. Springer (2021). https://doi.org/10.1007/978-3-030-74432-8_2
16. Girotto, S., Pizzi, C., Comin, M.: MetaProb: accurate metagenomic reads binning based on probabilistic sequence signatures. Bioinformatics **32**(17), i567–i575 (2016)
17. Li, Y., Han, R., Bi, C., Li, M., Wang, S., Gao, X.: DeepSimulator: a deep simulator for Nanopore sequencing. Bioinformatics **34**(17), 2899–2908 (2018)
18. Bonizzoni, P., De Felice, C., Zaccagnino, R., Zizza, R.: Inverse Lyndon words and inverse Lyndon factorizations of words. Advances in Applied Mathematics **101**, 281–319 (2018)
19. Bonizzoni, P., De Felice, C., Zaccagnino, R., Zizza, R.: On the longest common prefix of suffixes in an inverse lyndon factorization and other properties. Theoretical Computer Science **862**, 24–41 (2021)