A Resource-frugal Probabilistic Dictionary and Applications in (Meta)Genomics

Camille Marchet[1], Antoine Limasset[1], Lucie Bittner[2], and Pierre Peterlongo[1]

[1] IRISA Inria Rennes Bretagne Atlantique, GenScale team
[2] Sorbonne Universités, Université Pierre et Marie Curie (UPMC), CNRS, Institut de Biologie Paris-Seine (IBPS), Evolution Paris Seine, F-75005 Paris, France
Corresponding author pierre.peterlongo@inria.fr

Abstract. Genomic and metagenomic fields, generating huge sets of short genomic sequences, brought their own share of high performance problems. To extract relevant pieces of information from the huge data sets generated by current sequencing techniques, one must rely on extremely scalable methods and solutions. Indexing billions of objects is a task considered too expensive while being a fundamental need in this field. In this paper we propose a straightforward indexing structure that scales to billions of element and we propose two direct applications in genomics and metagenomics.
We show that our proposal solves problem instances for which no other known solution scales up. We believe that many tools and applications could benefit from either the fundamental data structure we provide or from the applications developed from this structure.

Keywords: bioinformatics; sequences comparison; genomics; metagenomics; data structures; minimal perfect hash functions; indexing

Introduction

A genome or a chromosome can be seen as a word of millions characters long, written in a four letters (or *bases*) alphabet. Modern molecular genome biology relies on sequencing, where the information contained in a genome is chopped into small sequences (around one hundred bases), called reads. By providing millions of short genomic reads along with reasonable sequencing costs, high-throughput sequencing technologies [31] (HTS) introduced an era where data generation is no longer a bottleneck while data analysis is, as this amount of sequences needs to be pulled together in a coherent way. Thanks to HTS improvements, it is possible to sequence hundreds of single genomes and RNA molecules, giving insight to diversity and expression of the genes. HTS even allow to go beyond the study of an individual by sequencing different species/organisms from the same environment at once, going from genomics to metagenomics. This massive sequencing represents a breakthrough: for instance one now can access and directly investigate the majority of the microbial world, which cannot be grown in the lab [17]. However, because of the diversity and complexity of microbial communities, such experiments produce tremendous volumes of data, which represent a challenge for bioinformaticians to deal with. The fragmented nature of genomic information, shredded in reads, craves algorithms to organize and make sense of the data.

A fundamental algorithmic need is to be able to index read sets for a fast information retrieval. In particular, given the amount of data an experiment can produce, methods that scale up to large data sets are needed. In this paper we propose a novel indexation method, called the quasi-dictionary, a probabilistic data structure based

Camille Marchet, Antoine Limasset, Lucie Bittner, Pierre Peterlongo: A Resource-frugal Probabilistic Dictionary and Applications in (Meta)Genomics, pp. 85–98.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8

on Minimal Perfect Hash Functions (MPHF). This technique provides a way to associate any kind of data to any piece of sequence from a read set, scaling to very large (billions of elements) data sets, with a low and controlled false positive rate.

A number of studies have focused on optimizing non-probabilistic text indexation, using for instance FM-index [13], or hash tables. However, except the Bloomier filter [9], to the best of our knowledge, no probabilistic dictionary has yet been proposed for which the false positive or wrong answer rates are mastered and limited. The quasi-dictionary mimics the Bloomier filter solution as it enables to associate a value to each element from a set, and to obtain the value of an element with a mastered false positive probability if the element was not indexed. Existing published results in [9] indicate that the Bloomier filter and the quasi-dictionary have similar execution times, while our results tend to show that the quasi-dictionary uses approximately ten times less memory. Moreover, there are no available/free Bloomier filter yet implemented.

We propose two applications that use quasi-dictionary for indexing k-mers, enabling to scale up large (meta)genomic instances. As suggested by their names (*short read connector counter* and *short read connector linker*, as presented below), these applications have the ability to connect any read to either its estimated abundance in any read set or to a list of reads in any read set. A key point of these applications is to estimate read similarity using k-mers diversity only. This alignment-free approach is widely used and is a good estimation of similarity measure [12].

Our first application, called *short read connector counter* (SRC_counter), consists in estimating the number of occurrences of a read (i.e. its abundance) in a read set. This is a central point in high-throughput sequencing studies. Abundance is first very commonly used as indicator value for reads trimming: i.e. reads with relatively low abundance value are considered as amplification errors and/or sequencing errors, and these rare reads are generally removed before thorough analyses [21,30]. The abundance of reads is then interpreted as a quantitative or semi-quantitative metric: i.e. reads abundance is used as a measure of genic or taxon abundance, themselves very commonly used for comparisons of community similarity [2,19].

The second proposed application in this work, called *short read connector linker* (SRC_linker), consists in providing a list of similar reads between read sets. We define the read set similarity problem as follows. Given a read set *bank* and a read set *query*, provide a similarity measure between each pair of reads $b_i \times q_j$, with b_i a read from the bank set and q_j a read from the query set. Note that the bank and the query sets may refer to the same data set. Computing read similarity intra-read set or inter-read sets can be performed by a general purpose tool, such as those computing similarities using dynamic programming, and using heuristic tools such as BLAST [1]. However, comparing all versus all reads requires a quadratic number of read comparisons, leading to prohibitive computation time, as this is shown in our proposed results. There exist tools dedicated to the computation of distances between read sets [7,25,26], but none of them can provide similarity between each pair of reads $b_i \times q_j$. Otherwise, some tools such as starcode [35] are optimized for pairwise sequence comparisons with mainly the aim of clustering DNA barcodes. As shown in results, such tools also suffer from quadratic computation time complexity and thus do not scale up data sets composed of numerous reads.

Availability and license Our proposed tools SRC_counter and SRC_linker were developed using the GATB library [11]. They may be used as stand alone tools or as li-

braries. They are licensed under the GNU Affero General Public License version 3 and can be downloaded from `http://github.com/GATB/short_read_connector`. Also licensed under the GNU Affero General Public License version 3, the quasi-dictionary can be downloaded from `http://github.com/pierrepeterlongo/quasi_dictionary`.

1 Methods

We first recall basic notations: a k-mer is a word of length k on an alphabet Σ. Given a read set[1] $\mathcal{R}$, a k-mer is said *solid* in $\mathcal{R}$ with respect to a threshold t if its number of occurrences in $\mathcal{R}$ is bigger or equal to t. Let $|w|$ denote the length of a word $w \in \Sigma^*$ and $|\mathcal{R}|$ denote the number of elements contained in $\mathcal{R}$.

1.1 Quasi-dictionary index

In the following, we present our indexing solution. Based on this solution, two applications are proposed sections 1.3 and 1.4.

The index we propose associates each solid k-mer from a read set $\mathcal{R}$ to a unique value in $[0, N-1]$, with N being the total number of solid k-mers in $\mathcal{R}$. Ideally, when querying a non indexed k-mer (i.e. a non solid k-mer or a k-mer absent from $\mathcal{R}$) the index returns no value. In our proposal, a non indexed k-mer may be associated to a value in $[0, N-1]$ with a probability $p > 0$. This is why we refer to our index as the ***quasi**-dictionary*, since it is a probabilistic index. However, note that querying any indexed k-mer provides a unique and deterministic answer.

We define the quasi-dictionary as follows :

Given a static set composed of N distinct elements, a quasi-dictionary is composed of two structures: a minimal perfect hash function MPHF (see for instance [5]) and a table of fingerprints $FingerPrints$.

The MPHF for S is a function such that:

$$\forall e \in S,\ \mathrm{MPHF}(e) = i \in [0, N-1]$$
$$\forall e_1, e_2 \in S,\ (\mathrm{MPHF}(e_1) = \mathrm{MPHF}(e_2)) \Leftrightarrow (e_1 = e_2)$$

The fingerprint table for S is composed of N elements. It assigns to each element form S an integer value in $[0, 2^f - 1]$, with f the size of the fingerprint in bits. This table is used to verify the membership of an element e to the indexed set of elements using the MPHF. When S represents a set of k-mers, the fingerprint table uses $f \leq 2k$ since a k-mer can be coded as a word of $2k$ bits. The false positive rate is then

$$\frac{2^{(2k-f)} - 1}{2^{2k}} \approx \frac{1}{2^f}.$$

1.2 Indexing solid *k*-mers using a quasi-dictionary

Algorithm 1 presents the construction of the quasi-dictionary. The set of solid k-mers (algorithm 1, line 1) is obtained using the DSK [28] method. The MPHF (algorithm 1, line 2) is computed using the MPHF library[2].

[1] Note that formally $\mathcal{R}$ should be denoted as a *"collection"* instead of a *"set"* as a read may appear twice or more in $\mathcal{R}$. However, to make the reading easier, we use in this manuscript the term "*set*" usually employed for describing HTS outputs.

[2] `https://github.com/rizkg/BooPHF`, commit number 852*cda*2

Algorithm 1: *create_quasidictionary*

Data: Read set $\mathcal{R}$, $k \in \mathbb{N}, t \in \mathbb{N}, f \in \mathbb{N}$
Result: A quasi-dictionary QD
1 k-mer set $\mathcal{K}$ = get_solid_kmers($\mathcal{R}, k, t$) ;
2 $QD.MPHF = create\_MPHF(\mathcal{K})$;
3 **foreach** *k-mer* w *in* $\mathcal{K}$ **do**
4 $\quad index = QD.MPHF(w)$;
5 $\quad QD.FingerPrints[index] = create\_fingerprint(w, f)$;
6 **return** QD;

The fingerprint of a word w (algorithm 1, line 5) is obtained thanks to a hashing function

$$create\_fingerprint : \Sigma^{|w|} \rightarrow [0, 2^f - 1],$$

with $f \leq 2k$. In practice we chose to use a xor-shift [27] for its efficiency in terms of throughput and hash distribution.

Algorithm 2: *query_quasidictionary*

Data: Quasi-dictionary QD, word w
Result: A unique value in $[0, N-1]$ (with N the number of indexed elements) or -1 if w detected as non indexed
1 $index = QD.MPHF(w)$;
2 **if** $index \geq 0$ ***and*** $QD.FingerPrints[index] = create\_fingerprint(w)$ **then**
3 $\quad$ **return** $index$;
4 **return** -1;

The querying of a quasi-dictionary with a word w is straightforward, as presented in Algorithm 2. The *index* of w is retrieved using the MPHF. Then the fingerprint stored for this *index* is compared to the fingerprint of w. If they differ, then w is not indexed and the -1 value is returned. If they are equal, the value $index \geq 0$ is returned. Note that two distinct words have the same fingerprint with a probability $\approx \frac{1}{2^f}$. It follows that there is a probability $\approx \frac{1}{2^f}$ that the quasi-dictionary returns a false positive value despite the fingerprint checking, *i.e.* an $index \neq -1$ for a non indexed word. On the other hand, the *index* returned for an indexed word is the correct one. In practice we use $f = 12$ that limits the false positive rate to $\approx 0.02\%$. Note that our implementation authorizes any value $f \leq 64$.

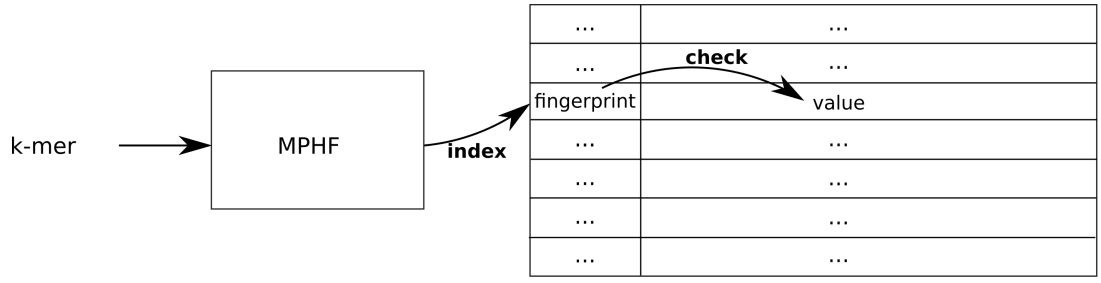

Figure 1. Quasi dictionary structure composed of a MPHF and a table of fingerprints. The fingerprints are obtained by hashing the corresponding k-mers. Thanks to the MPHF a unique index is associated to each k-mer where the fingerprint is stored. The fingerprint is associated to the value we want to link to the k-mer. When a k-mer is queried the fingerprint obtained is checked against the fingerprint stored to limit false positives.

DNA strands DNA molecules are composed of two strands, each one being the reverse complement[3] of the other. As current sequencers usually do not provide the strand of each sequenced read, each indexed or queried k-mer should be considered in the forward or in the reverse complement strand. This is why, in the proposed implementations, we index and query only the canonical representation of each k-mer, which is the lexicographically smaller word between a k-mer and its reverse complement.

Time and memory complexities Our MPHF implementation has the following characteristics. The structure can be constructed in $O(N)$ time and uses ≈ 4 bits by elements. We could use parameters limiting memory fingerprint to less than 3 bits per element, but we chose parameters to greatly speed up MPHF construction and query. The fingerprint table is constructed in $O(N)$ time, as the $create\_fingerprint$ function runs in $O(1)$. This table uses exactly $N \times f$ bits. Thus the overall quasi-dictionary size, with $f = 12$ is ≈ 16 bits per element. Note that this value does not take into account the size of the values associated to each indexed element.

The querying of an element is performed in constant time and does not increase memory complexity.

1.3 Approximating the number of occurrences of a read in a read set

Algorithm 3: SRC_counter: Quasi-dictionary used for counting k-mers

```
   Data: Read set B, read set Q, k ∈ ℕ, t ∈ ℕ, f ∈ ℕ
   Result: For each read from Q, its k-mer similarity with set B
1  quasi-dictionary QD = create_quasidictionary(B, k, t, f) ;
2  create a table count composed of N integers[a];
3  foreach Solid k-mer w from B do
4  |  count[query_quasidictionary(w)] = number of occurrences of w in B;
5  foreach read q in Q do
6  |  create an empty vector count_q;
7  |  foreach k-mer w in q do
8  |  |  if query_quasidictionary(w) ≥ 0 then
9  |  |  |  add count[query_quasidictionary(w)] to count_q ;
10 |  Output the q identifier, and (mean, median, min and max values of count_q);
```

[a] with N the number of solid k-mers from $\mathcal{B}$

As presented in Algorithm 3, we propose a first straightforward application using the quasi-dictionary. This application is called SRC_counter for *short read connector counter*. It approximates the number of occurrences of reads in a read set.

Two potentially equal read sets $\mathcal{B}$ and $\mathcal{Q}$ are considered. The indexation phase works as follows. Each solid k-mer of $\mathcal{B}$ is indexed using a quasi-dictionary. A third-party table named *count* stores the counts of indexed k-mers. Elements of this table are accessed via the quasi-dictionary *index* value of indexed items (Algorithm 3 lines 4 and 9). The number of occurrences of each solid k-mer from $\mathcal{B}$ (line 4) is obtained from DSK output, used during the quasi-dictionary creation (line 1). Then starts the query phase. Once the *count* table is created, for each read q from set $\mathcal{Q}$, the count of all its k-mers indexed in the quasi-dictionary are recovered and stored in a vector

[3] The reverse complement of a DNA sequence is the palindrome of the sequence, in which A and T are swapped and C and G are swapped. For instance the reverse complement of $ACCG$ is $CGGT$.

(lines 8 and 9). Finally, collected counts from k-mers from q are used to output an estimation of its abundance in read set $\mathcal{B}$. The abundance is approximated using the mean number of occurrences of k-mers from q, to supplement we output the median, the min and the max number of occurrences of k-mers from q. In rare occasions, false positives of the method can lead to an over-estimation of the count.

This algorithm is extremely simple. In addition to the quasi-dictionary creation time and memory complexities, it has a constant memory overhead (8 bits by element in our implementation) and it has an additional $O(\sum_{Q\in\mathcal{Q}} |Q|)$ time complexity.

1.4 Identifying similar reads between read sets or inside a read set

Algorithm 4: SRC_linker: Quasi-dictionary used for identifying read similarities

```
   Data: Read set B, read set Q, k ∈ ℕ, t ∈ ℕ, f ∈ ℕ
   Result: For each read from Q, its k-mer similarity with each read from set B
 1 quasi-dictionary QD = create_quasidictionary(B, k, t, f) ;
 2 create a table ids composed of N vectors of integers[a] ;
 3 foreach read b in B do
 4 |   foreach k-mer w in b do
 5 |   |   index = query_quasidictionary(w);
 6 |   |   if index ≥ 0 then
 7 |   |   |   add identifier of b to vector ids[index] ;
 8 foreach read q in Q do
 9 |   create a hash table associating targets (target_read_id) to couple(next_free_position,
   |   count);
10 |   foreach i in [0, |q| − k] do
11 |   |   w = k-mer occurring position i in q;
12 |   |   index = query_quasidictionary(w);
13 |   |   if index ≥ 0 then
14 |   |   |   foreach tg_id in vector ids[index] do
15 |   |   |   |   if targets[tg_id] is empty then
16 |   |   |   |   |   targets[tg_id].next_free_position = 0
17 |   |   |   |   targets[tg_id].count+ = max (k, i + k − targets[tg_id].next_free_position)
   |   |   |   |   targets[tg_id].next_free_position = i + k
18 |   Output the id of q and each[b] tg_id associate to its count from targets table;
```

[a] with N the number of solid k-mers from $\mathcal{B}$

[b] In practice only *tg_id* whose *count* value is higher or equal to a user defined threshold are output

Our second proposal, called SRC_linker for *short read connector linker*, compares reads from two potentially identical read sets $\mathcal{B}$ and $\mathcal{Q}$. For each read q from $\mathcal{Q}$, a similarity measure with reads from $\mathcal{B}$ is provided.

The similarity measure we propose for a couple of reads $q \times b$ is the number of positions on q that is covered by at least a k-mer that also occur on b. Note that this measure is not symmetrical as one does not verify that the k-mers do not overlap on b.

The indexation phase of SRC_linker works as follows. A quasi-dictionary is created and a third-party table *ids* of size N is created. Each element of this table stores for a solid k-mer w from $\mathcal{B}$ a vector containing the identifiers of reads from $\mathcal{B}$ in which w occurs. See lines 2 to 7 of Algorithm 4.

The query phase (lines 8 to the end of Algorithm 4) is straightforward. In practice, for each targeted read b_j in $\mathcal{B}$ we remind the ending position of the last shared k-mer

on q with b_j denoted by $next\_free\_position$ in the Algorithm 4. Given a new shared k-mer, the number of positions that was not already covered by another shared k-mer is added to the similarity measure (line 17 of Algorithm 4).

Once all k-mers of a read q are treated, the identifier of q is output and for each read b_j from $\mathcal{B}$ its identifier is output together with the number of shared k-mers with q. In practice, in order to avoid quadratic output size and to focus on similar reads, only reads sharing a number of k-mers higher or equal to a user defined threshold are output.

In addition to the quasi-dictionary data structure creation, considering a fixed read size, Algorithm 4 has $O(N \times \overline{m})$ memory complexity and a $O(N + \sum_{Q \in \mathcal{Q}} |Q| \times \overline{m})$ time complexity, with $\overline{m}$ the average number of distinct reads from $\mathcal{B}$ in which a k-mer from $\mathcal{Q}$ occurs. In the worst case $\overline{m} = N$, for instance with $\mathcal{B} = \mathcal{Q} = \left\{A^{|\text{read}|}\right\}^N$. In practice, in our tests as well as for real sets composed of hundred of million reads, $\overline{m}$ is limited to ≈ 2.22.

Storing read identifiers on disk Storing the read identifiers as proposed in Algorithm 4 presents important drawbacks as it requires a large amount of RAM. In order to get rid of this limitation we propose a disk version of this algorithm, in which the table *ids* is stored on disk. As shown in Algorithm 5 (see Appendix), the algorithmic solution is not straightforward as one needs to know for each indexed k-mer w its number of occurrences in the read set $\mathcal{B}$ plus the number of occurrences of k-mers $\neq w$ from $\mathcal{B}$ (false positives) that have the same quasi-dictionary *index*.

This disk based solution enables to scale up very large instances with frugal RAM needs, at the price of a longer computation time, as show in results.

2 Results

This section presents results about the fundamental quasi-dictionary data structure and about potential applications derived from its usage. To this end, we use a metagenomic *Tara* Oceans [18] read set ERR59928[4] composed of 189,207,003 reads of average size 97 nucleotides. From this read set, we created six sub-sets by selecting first 10K, 100K, 1M, 10M, 50M and 100M reads (with K meaning thousand and M meaning million).

Tests were performed on a linux 20-CPU nodes running at 2.60 GHz with an overall of 252 GBytes memory.

2.1 SRC_counter tests and performances

We first provide SRC_counter results enabling to evaluate the gain of our proposed data structure when compared to a classical hash table. Secondly we provide results that enable to estimate the impact of false positives on results.

SRC_counter performances compared to standard hash table index We tested the SRC_counter performances by indexing iteratively the six read subsets plus the full ERR59928 set, each time querying reads from set 10M. We compared our solution performances with a classical indexation scheme done using the C++

[4] `http://www.ebi.ac.uk/ena/data/view/ERR599280`

Indexed Dataset (nb solid k-mers)	k-mer count time (s)	Construc. time (s)		Memory (GB)			Query Time(s)	
		QD	Hash	QD	QD62	Hash	QD	Hash
1M (64,321,167)	2	1	106	0.25	2.45	2.46	10	13
10M (621,663,812)	15	7	1091	1.80	5.45	23.58	11	17
50M (2,812,637,134)	72	77	5027	8.00	16.37	106.25	11	19
100M (5,191,190,377)	196	220	9335	14.71	44.93	202.91	13	19
Full (8,783,654,120)	486	532		24.83	75.96		15	

Table 1. Wallclock time and memory used by the SRC_counter algorithm for creating and for querying the quasi-dictionary using the default fingerprint size $f = 8$ (denoted by "QD") and the C++ *unordered_map*, denoted by "Hash". Column "k-mer count time" indicates the time DSK spent counting k-mers. Tests were performed using $k = 31$ and $t = 1$ (all k-mers are solid). The query read set was always the 10M set. We additionally provide memory results using the quasi-dictionary with a fingerprint size $f = 62$ (denoted by "QD62"). Construction and query time for QD62 are not shown as they are almost identical to the QD ones. On the full data set, using a classical hash table, the memory exceeded the maximal authorized machine limits (252 GB).

unordered_map hash table. Results are presented in Table 1. These results show that the quasi-dictionary is much faster to compute than this hash table solution, in particular because of parallelisation. Moreover, the quasi-dictionary memory footprint is ≈ 13 times smaller on large enough instances (10 million indexed reads or more). These results show that the hash table is not a viable solution scaling up current read sets composed of several billions k-mers. Results also highlight the fact that the query is fast and only slightly depends on the number of indexed elements.

Importantly, using a fingerprint large enough (here $f = 62$ for k-mers of length $k = 31$), we can force the quasi-dictionary to avoid false positives. As expected, the quasi-dictionary data structure size increases with the size of f but interestingly, on this example and as shown in Table 1, the size of the quasi-dictionary with $f = 62$ remains in average 4 times smaller than the size of the hash-table on large problem instances. Keeping in mind that the quasi-dictionary is faster to construct and to query, the usage of this data structure avoiding false positives presents only advantages compared to the hash table usage for indexing a static set. However, one should recall that this is true because we are using an alphabet of size four, so any 31-mer on the alphabet $\{A, C, G, T\}$ can be assigned to a unique value in $[0, 2^{62} - 1]$ and *vice versa*. With larger alphabets such as the amino-acids or the Latin ones, the usage of a hash table is recommended if false positives are not tolerated.

Approximating false positives impact We propose an experiment to assess the impact on result quality when using a probabilistic data structure instead of a deterministic one for estimating read abundances.

We used the read set 100M both for the indexation and for the querying, thus providing an estimation of the abundance of each read in its own read set. We made the indexation using $k = 31, c = 2$ and $f = 8$. Note that, with $c = 2$ only k-mers seen twice or more in the set are solid and thus are indexed. In this example only 756,804,245 k-mers are solid among the 5,191,190,377 distinct k-mers present in the read set. This means that during the query, 85.4% of queried k-mers are not indexed. This enables to measure the impact of the quasi-dictionary false positives. We applied the count algorithm as described in Algorithm 3, and the tuned version using a hash table instead of a quasi-dictionary. We analyzed the count output composed of the average number of occurrences of k-mers of each read in the 100M read set.

Because of the quasi-dictionary false positives, results obtained using this structure are an over-estimation of the real result. Thus, we computed for each read the observed difference in the counts between results obtained using the quasi-dictionary implementation and the hash table implementation. The max over-approximation is 26.9, and the mean observed over-approximation is 7.27×10^{-3} with a 3.59×10^{-3} standard deviation. Thus, as the average estimated abundance of each read which is ≈ 2.22, the average count over-estimation represents $\approx 0.033\%$ of this value. Such divergences are negligible.

2.2 Identifying similar reads

We set a benchmark of our method with comparisons to state of the art tools that can be used in current pipelines for the read similarity identification presented in this paper. We compared our tool with the classical method BLAST [1] (version 2.3.0), with default parameters. BLAST is able to index big data sets, and consumes a reasonable quantity of memory, but the throughput of the tool is relatively low and only small data sets were treated within the timeout (10h, wallclock time). We also included two broadly used mappers in the comparison. We used Bowtie2 [22] (version 2.2.7), and BWA [23] (version 0.7.10) in any alignment mode (`-a` mode in Bowtie2, `-N` for BWA) in order to output all alignment found instead of the best ones only. Both tools are not well suited to index large set of short sequences nor to find all alignments and therefore use considerably more resources than their standard usage.

We also compared SRC_linker to starcode (1.0), that clusters DNA sequences by finding all sequences pairs below a Levenshtein distance metric. One should notice that benchmark comparisons with tools as starcode is unfair as such tool provides much more precise distance information between pair of reads than SRC_linker and performs additional task as clustering. However, our benchmark highlights the fact that such approaches suffer from intractable number of read comparisons, as demonstrated by presented results.

	Time(s)					Memory(GB)				
Indexed Dataset	Blast	Bowtie2	BWA	starcode	SRC_linker	Blast	Bowtie2	BWA	starcode	SRC_linker
10K	4	3	6	2	1	0.7	0.29	0.04	11.36	1.01
100K	52	51	106	29	5	18.5	0.77	0.49	12.06	1.07
1M	**795**	**10,644**	**3,155**	**1,103**	**45**	**24.5**	**5.54**	**3.4**	**18.18**	**1.28**
10M			62,912	131,139	587			5.9	73.5	3.61
100M					14,748					44.37
Full					40,828					110.84

Table 2. CPU time and memory consumption for indexing and querying a data set versus itself. We set a timeout of 10h. BLAST crashed for 10M data set, Bowtie2 reached the timeout we set with more than 200h (CPU) for 10M reads. BWA performs best among the mappers, reaching the timeout for 100M reads (more than 200h (CPU) on this data set). On the 100M data set, starcode reached the timeout. Only SRC_linker finished on all data sets. On the full data set, it lasted an order of magnitude comparable to what BWA performed on only 10M.

We focused on a practical use case for which our method could be used, namely retrieving similarities in a read set against itself. We used default SRC_linker parameters ($k = 31$, $f = 12$, $c = 2$). Because of the limitations of the methods we used for the benchmark, reported in Table 2, we could compare against all methods only up to 1M reads. BWA performed better than the two other tools in terms of memory,

being able to scale up to 10M reads, while Bowtie2 and BLAST could only reach 1M reads comparison. On this modest size of read set, we show that we are already ahead both in terms of memory and time. However the gap between our approach and others increases with the amount of data to process. Dealing with the full Tara data set reveals the specificity of our approach (Table 2) that requires low resources in comparison to others and is able to deal with bigger data sets.

	Indexation Time (s)		Query Time (s)		Memory
	One thread	20 threads	One thread	20 threads	(GB)
RAM Full	18,067	1,768	17,558	992	110
Disk Full	106,766	28,471	24,873	1,736	19

Table 3. Multithreading and disk performances. The full read set was used to detail the performances of the RAM and Disk algorithm on a large data set. We used default parameters $k = 31$, $f = 12$, $c = 2$. Times are wallclock times.

Finally, we highlight that we provide a parallelised tool (10× speedup for the index and 17× speedup for query for RAM algorithm as shown in Table 3) on the contrary to classical methods that are partly-parallelised as only the alignment step is well suited for parallelisation. The disk version does not fully benefit from multiple cores since the bottleneck is disk access. The main interest of this technique is a highly reduced memory usage at the price of an order of magnitude lower throughput, as presented Table 3.

3 Discussions and conclusion

In this contribution, we propose a new indexation scheme based on a Minimal Perfect Hash Function (MPHF) together with a fingerprint value associated to each indexed element. Our proposal is a probabilistic data structure that has similar features than Bloomier filters, with smaller memory fingerprint. This solution is resource-frugal (we have shown experiments on sets containing more than eight billion elements indexed in ≈ 3 hours and using less than 25 GB RAM) and opens the way to new (meta)genomic applications. As proofs of concept, we proposed two novel applications: SRC_counter and SRC_linker. The first estimates the abundance of a sequence in a read set. The second detects similarities between pair of reads inter or intra-read sets. These applications are a start for broader uses and purposes.

Two main limitations of our proposal due to the nature of the data structure can be pointed out. Firstly, compared to standard hash tables, our indexing data structure presents an important drawback: the exact set of keys to index has to be defined during the data structure creation and it has to be static. This may be a limitation for non fixed set of keys. Moreover, our data structure can generate false positives during query. Even with the proposed false positive ratio limited to $\approx 10^{-2}\%$ with defaults parameters, this may be incompatible with some applications. However we can force our tools to avoid false positives by using as a fingerprint the key itself. Interestingly, this still provides better time and memory performances than using a standard hash table in the DNA k-mer indexing context, with $k = 31$, which is a very common value for read comparisons [7]. Secondly, one should notice that our indexation proposal saves space regarding the association between an element and a specific array offset (if the element was indexed). However, our proposal does not

limit the space needed for storing the value associated to each indexed element. Thus, with respect to classical hash tables, the memory gain is limited in problem instances in which large values are associated to each key. Indeed, in this case, the memory footprint is mainly due to the value over the indexing scheme. In order to benefit from our proposal even in such cases we proposed an application example in which the values are stored on disk. However, our approach is namely designed for problems where a huge number of elements to index are at stake, along with a small quantity of information to match with.

We could improve our technique to recognize key from the original set, using a technique from the hashing field [20] or from the set representation field [6]. In such framework, a set can be represented with less memory than the sum of the memory required by the keys. We could thus hope being able to represent a non-probabilistic dictionary without storing keys. Otherwise, we could use the hashing information to achieve a smaller false positive rate with the same or a reduced memory usage. The main challenge will be to keep fast query operation for such complex data structure.

The results we provided show that alignment-based approaches do not scale when it comes to find similar reads in data sets composed of millions of sequences. The fact that HTS data count rarely less than millions reads justifies our approach based on k-mer similarity. Moreover our approach is more straightforward and requires less parameters and heuristics than mapping approaches, that can sometimes turn them into blackboxes. However, such an approach remains less precise than mapping, since the k-mer order is not taken into account and is less sensitive because of the fixed size of k. An important future work will be to evaluate the differences between matches of our pseudo-alignment and matches of well-known and widely used tool as BLAST.

Our tools property of enabling the test of a read set against itself opens the door to applications such as read clustering. Latest sequencing technologies, called Third Generation Sequencers (TGS), provide longer reads [32,33] (more than a thousand bases instead of a few hundreds for HTS). With previous HTS short reads, *de novo* approaches to reconstruct DNA or RNA molecules were using assembly [16,29], based on de Bruijn graphs. For RNA, these TGS long reads mean a change of paradigm as assembly is no more necessary, as one read is long enough to represent one full-length molecule. The important matter becomes to segregate families of RNA molecules within a read set, a purpose our approach could be designed for.

Furthermore, the methods we provide have straightforward applications examples in biology, such as the building of sequences similarity networks (SSN) [3] using SRC_linker. SSN are extremely useful for biologists because, in addition to allowing a user-friendly visualization of the genetic diversity from huge HTS data sets, they can be studied analytically and statistically using graph topology metrics. SSN have recently been adapted to address an increasing number of biological questions investigating both patterns and processes: e.g. population structuring [15,14]; genomes heterogeneity [8]; microbial complexity and evolution [10]; microbiome adaptation [4,34] or to explore the microbial dark matter [24]. In metagenomic microbial studies, SSN offer an alternative to classical and potentially biased methods, and thus facilitate large-scale analyses and hypotheses generation, while notably including unknown/dark matter sequences in the global analysis [15,24]. Currently SSN are built upon general purposes tools such as BLAST. They thus hardly scale up large data sets. A future work will consist in checking the feasibility of applying SRC_linker for constructing SSN and, in case of success, to use it on large SSN problem instances on which other classical tools cannot be applied.

Acknowledgments

This work was funded by French ANR-12-BS02-0008 Colib'read project. We thank the GenOuest BioInformatics Platform that provided the computing resources necessary for benchmarking. We warmly thank Guillaume Rizk and Rayan Chikhi for their work on the MPHF and for their feedback on the preliminary version of this manuscript.

References

1. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman: *Basic local alignment search tool.* Journal of molecular biology, 215(3) 1990, pp. 403–410.
2. A. S. Amend, K. A. Seifert, and T. D. Bruns: *Quantifying microbial communities with 454 pyrosequencing: does read abundance count?* Mol. Ecol., 19(24) Dec 2010, pp. 5555–5565.
3. H. J. Atkinson, J. H. Morris, T. E. Ferrin, and P. C. Babbitt: *Using sequence similarity networks for visualization of relationships across diverse protein superfamilies.* PLoS ONE, 4(2) 2009, p. e4345.
4. E. Bapteste, C. Bicep, and P. Lopez: *Evolution of genetic diversity using networks: the human gut microbiome as a case study.* Clin. Microbiol. Infect., 18 Suppl 4 Jul 2012, pp. 40–43.
5. D. Belazzougui, P. Boldi, G. Ottaviano, R. Venturini, and S. Vigna: *Cache-oblivious peeling of random hypergraphs*, in Data Compression Conference Proceedings, 2014, pp. 352–361.
6. D. Belazzougui and R. Venturini: *Compressed static functions with applications*, in Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '13, Philadelphia, PA, USA, 2013, Society for Industrial and Applied Mathematics, pp. 229–240.
7. G. Benoit, P. Peterlongo, M. Mariadassou, E. Drezen, S. Schbath, D. Lavenier, and C. Lemaitre: *Multiple Comparative Metagenomics using Multiset k-mer Counting.* apr 2016, pp. 1–17.
8. E. Boon, S. Halary, E. Bapteste, and M. Hijri: *Studying genome heterogeneity within the arbuscular mycorrhizal fungal cytoplasm.* Genome Biol Evol, 7(2) Feb 2015, pp. 505–521.
9. D. Charles and K. Chellapilla: *Bloomier filters: A second look*, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 5193 LNCS, 2008, pp. 259–270.
10. E. Corel, P. Lopez, R. Meheust, and E. Bapteste: *Network-Thinking: Graphs to Analyze Microbial Complexity and Evolution.* Trends Microbiol., 24(3) Mar 2016, pp. 224–237.
11. E. Drezen, G. Rizk, R. Chikhi, C. Deltel, C. Lemaitre, P. Peterlongo, and D. Lavenier: *GATB: Genome Assembly & Analysis Tool Box.* Bioinformatics (Oxford, England), jul 2014, pp. 1–3.
12. V. B. Dubinkina, D. S. Ischenko, V. I. Ulyantsev, A. V. Tyakht, and D. G. Alexeev: *Assessment of k-mer spectrum applicability for metagenomic dissimilarity analysis.* BMC Bioinformatics, 17(1) dec 2016, p. 38.
13. P. Ferragina and G. Manzini: *Indexing Compressed Text.* Journal of the ACM, 52(4) 2000, pp. 552–581.
14. M. Fondi, A. Karkman, M. Tamminen, E. Bosi, M. Virta, R. Fani, E. Alm, and J. McInerney: *Every gene is everywhere but the environment selects: Global geo-localization of gene sharing in environmental samples through network analysis.* Genome Biology and Evolution, 2016.
15. D. Forster, L. Bittner, S. Karkar, M. Dunthorn, S. Romac, S. Audic, P. Lopez, T. Stoeck, and E. Bapteste: *Testing ecological theories with sequence similarity networks: marine ciliates exhibit similar geographic dispersal patterns as multicellular organisms.* BMC Biol., 13 2015, p. 16.
16. M. G. Grabherr, B. J. Haas, M. Yassour, J. Z. Levin, D. A. Thompson, I. Amit, X. Adiconis, L. Fan, R. Raychowdhury, Q. Zeng, et al.: *Full-length transcriptome assembly from RNA-Seq data without a reference genome.* Nature biotechnology, 29(7) 2011, pp. 644–652.
17. L. A. Hug, B. J. Baker, K. Anantharaman, C. T. Brown, A. J. Probst, C. J. Castelle, C. N. Butterfield, A. W. Hernsdorf, Y. Amano, K. Ise, Y. Suzuki, N. Dudek, D. A. Relman, K. M. Finstad, R. Amundson, B. C. Thomas, and J. F. Banfield: *A new view of the tree of life.* Nature Microbiology, 1 Apr 2016, pp. 16048 EP –, Letter.

18. E. Karsenti, S. G. Acinas, P. Bork, C. Bowler, C. de Vargas, J. Raes, M. Sullivan, D. Arendt, F. Benzoni, J. M. Claverie, M. Follows, G. Gorsky, P. Hingamp, D. Iudicone, O. Jaillon, S. Kandels-Lewis, U. Krzic, F. Not, H. Ogata, S. Pesant, E. G. Reynaud, C. Sardet, M. E. Sieracki, S. Speich, D. Velayoudon, J. Weissenbach, and P. Wincker: *A holistic approach to marine Eco-systems biology.* PLoS Biology, 9 2011.
19. S. W. Kembel, M. Wu, J. A. Eisen, and J. L. Green: *Incorporating 16s gene copy number information improves estimates of microbial diversity and abundance.* PLoS Comput Biol, 8(10) 10 2012, pp. 1–11.
20. A. Kirsch and M. Mitzenmacher: *Less hashing, same performance: Building a better Bloom filter*, in Algorithms–ESA 2006, Springer, 2006, pp. 456–467.
21. V. Kunin, A. Engelbrektson, H. Ochman, and P. Hugenholtz: *Wrinkles in the rare biosphere: pyrosequencing errors can lead to artificial inflation of diversity estimates.* Environ. Microbiol., 12(1) Jan 2010, pp. 118–123.
22. B. Langmead and S. L. Salzberg: *Fast gapped-read alignment with Bowtie 2.* Nature methods, 9(4) 2012, pp. 357–359.
23. H. Li and R. Durbin: *Fast and accurate short read alignment with Burrows–Wheeler transform.* Bioinformatics, 25(14) 2009, pp. 1754–1760.
24. P. Lopez, S. Halary, and E. Bapteste: *Highly divergent ancient gene families in metagenomic samples are compatible with additional divisions of life.* Biol. Direct, 10 2015, p. 64.
25. N. Maillet, G. Collet, T. Vannier, D. Lavenier, and P. Peterlongo: *COMMET: comparing and combining multiple metagenomic datasets*, in Bioinformatics and Biomedicine (BIBM), 2014 IEEE International Conference on, IEEE, 2014, pp. 94–98.
26. N. Maillet, C. Lemaitre, R. Chikhi, D. Lavenier, and P. Peterlongo: *Compareads: comparing huge metagenomic experiments.* BMC Bioinformatics, 13(19) 2012, pp. 1–10.
27. G. Marsaglia: *Xorshift rngs.* Journal of Statistical Software, 8(14) 2003, pp. 1–6.
28. G. Rizk, D. Lavenier, and R. Chikhi: *DSK: K-mer counting with very low memory usage.* Bioinformatics, 29(5) 2013, pp. 652–653.
29. G. Robertson, J. Schein, R. Chiu, R. Corbett, M. Field, S. D. Jackman, K. Mungall, S. Lee, H. M. Okada, J. Q. Qian, et al.: *De novo assembly and analysis of RNA-seq data.* Nature methods, 7(11) 2010, pp. 909–912.
30. M. Schirmer, U. Z. Ijaz, R. D'Amore, N. Hall, W. T. Sloan, and C. Quince: *Insight into biases and sequencing errors for amplicon sequencing with the illumina miseq platform.* Nucleic Acids Research, 2015.
31. S. C. Schuster: *Next-generation sequencing transforms todays biology.* Nature, 200(8) 2007, pp. 16–18.
32. D. Sharon, H. Tilgner, F. Grubert, and M. Snyder: *A single-molecule long-read survey of the human transcriptome.* Nature biotechnology, 31(11) 2013, pp. 1009–1014.
33. H. Tilgner, F. Grubert, D. Sharon, and M. P. Snyder: *Defining a personal, allele-specific, and single-molecule long-read transcriptome.* Proceedings of the National Academy of Sciences, 111(27) 2014, pp. 9869–9874.
34. F. Völkel, E. Bapteste, M. Habib, P. Lopez, and C. Vigliotti: *Read networks and k-laminar graphs.* arXiv, 2016, pp. 1–14.
35. E. Zorita, P. Cuscó, and G. J. Filion: *Starcode: sequence clustering based on all-pairs search.* Bioinformatics, 31(12) jun 2015, pp. 1913–1919.

4 Appendix

Appendix contains a presentation of the SRC_linker algorithm using disk for storing values (Algorithm 5).

Algorithm 5: SRC_linker_Disk: Quasi-dictionary used for identifying read similarities

Data: Read set $\mathcal{B}$, read set $\mathcal{Q}$, $k \in \mathbb{N}, t \in \mathbb{N}, f \in \mathbb{N}$
Result: For each read from $\mathcal{Q}$, its k-mer similarity with each read from set $\mathcal{B}$

```
1  quasi-dictionary QD = create_quasidictionary(B, k, t, f) ;
2  create a table ids composed of N integers[a] all valued to 0;
3  foreach read b in B do
4      foreach k-mer w in b do
5          index = query_quasidictionary(w);
6          if index ≥ 0 then
7              add 1 to ids[index];
8  foreach Solid k-mer w from B do
9      index = query_quasidictionary(w);
10     if index ≥ 0 then
11         count = ids[index];
12         ids[index] = Temporary_File.position;
13         write count + 1 '0' on Temporary_File;
14 foreach read b in B do
15     foreach k-mer w in b do
16         index = query_quasidictionary(w);
17         if index ≥ 0 then
18             position = ids[index];
19             Temporary_File.goto(position);
20             write id of b in place of the first 0 found;
21 foreach read q in Q do
22     create a hash table targets (target_read_id) → couple(next_free_position, count);
23     foreach i in [0, |q| − k] do
24         w = k-mer occurring position i in q;
25         index = query_quasidictionary(w);
26         if index ≥ 0 then
27             position = ids[index];
28             Temporary_File.goto(position);
29             read from Temporary_File and put in vector V all integer until a 0 is found;
30             foreach tg_id in vector V do
31                 if targets[tg_id] is empty then
32                     targets[tg_id].next_free_position = 0
33                 targets[tg_id].count+ = max (k, i + k − targets[tg_id].next_free_position)
                   targets[tg_id].next_free_position = i + k
34     Output the id of q and each[b] tg_id associate to its count from targets table;
```

[a] with N the number of solid k-mers from $\mathcal{B}$
[b] In practice only *tg_id* whose *count* value is higher or equal to a user defined threshold are output