Compareads: comparing huge metagenomic experiments

Nicolas Maillet*[1], Claire Lemaitre[1], Rayan Chikhi[2], Dominique Lavenier[1], Pierre Peterlongo*[1]

[1]INRIA Rennes - Bretagne Atlantique / IRISA, EPI GenScale, Rennes, France
[2]ENS Cachan/IRISA, EPI Symbiose, Rennes, France

Email: Nicolas Maillet - nicolas.maillet@inria.fr; Claire Lemaitre - claire.lemaitre@inria.fr; Rayan Chikhi - rayan.chikhi@irisa.fr; Dominique Lavenier - dominique.lavenier@irisa.fr; Pierre Peterlongo*- pierre.peterlongo@inria.fr;

*Corresponding authors

Abstract

Background: Nowadays, metagenomic sample analyses are mainly achieved by comparing them with *a priori* knowledge stored in data banks. Even if powerful, such approaches do not allow to exploit unknown and/or "unculturable" species, for instance estimated at 99% for Bacteria.

Methods: This work introduces Compareads, a *de novo* comparative metagenomic approach that returns the reads that are similar between two possibly metagenomic datasets generated by High Throughput Sequencers. One originality of this work consists in its ability to deal with huge datasets. The second main contribution presented in this paper is the design of a probabilistic data structure based on Bloom filters enabling to index millions of reads with a limited memory footprint and a controlled error rate.

Results: We show that Compareads enables to retrieve biological information while being able to scale to huge datasets. Its time and memory features make Compareads usable on read sets each composed of more than 100 million Illumina reads in a few hours and consuming 4Gb of memory, and thus usable on today's personal computers.

Conclusion: Using a new data structure, Compareads is a practical solution for comparing *de novo* huge metagenomic samples. Compareads is released under the CeCILL license and can be freely downloaded from http://alcovna.genouest.org/compareads/.

Introduction

The past five years have seen the arrival of High Throughput Sequencing (HTS), also known as Next-Generation Sequencing (NGS). These technologies drastically lowered sequencing costs and increased sequencing throughput. They radically changed molecular biology and computational biology, as data generation is no longer a bottleneck. In fact, nowadays a major challenge is the analysis and interpretation of sequencing data [1]. HTS democratized access to sequencing to almost all biological labs over the world. It also opened the doors to new techniques such as ChipSeq [2], ClipSeq [3], RadSeq [4] and the topic of this work, metagenomics [5].

Metagenomics, also known as "environmental genomics", provides an alternative to traditional single-genome studies for exploring the microbial world. Most microorganisms (up to 99% of Bacteria [6]) are unknown and possibly "unculturable". Even if traditional genomics sequencing methods are well studied, they are not suited for environmental samples, because of the need to cultivate clones. By sequencing uncultured genomes directly from environmental samples, metagenomics offers new ways to study this unexplored diversity.

HTS technologies provide fragments of sequences (called reads) of length a few hundred base pairs without any information about the locus nor the orientation on the molecule they come from. In the metagenomic context, an additional difficulty comes from the fact that each read may belong to any species.

Nowadays, it is difficult to assemble complex metagenomes (such as soil or water metagenomes) into longer consensus sequences, because reads from different species may be merged into one chimeric sequence. Mende and colleagues [7] showed that for a 400-genomes metagenome, using simulated Illumina reads, 37% of the assembled sequences were chimeric. Thus currently, reads from metagenomes are used to estimate the biodiversity [8] or may be compared to known databases, providing information w.r.t. the current scientific knowledge [9, 10]. Another way to exploit two or more metagenomic datasets is to compare them together, enabling to understand how genomic differences are related to environmental ones (biotopes localizations and/or time spent after an event).

Comparative metagenomics can deal with many aspects, such as sequence composition, *i.e.* GC content [11], and genome size [12], taxonomic diversity [13], functional content [14], etc. Several methods are currently developed for comparative metagenomics analyses. Some are based on statistical methods to deal with such large number of descriptive variables, *e.g.* principal component analysis (PCA).

To the best of our knowledge, there is no software designed to compare two or more metagenomic samples at the read level, that is identifying reads that are shared or similar between samples. This can be simply used to compute a similarity measure between samples such as the number or purcentage of similar reads between pairs of samples. When dealing with more than two samples, this would enable among others to classify metagenomics samples based on their raw reads content. One could use the popular tool Blast to align reads in an all-vs-all way, however it is not designed specifically to this task and more importantly it can not cope in time and memory with the size of nowadays metagenomic samples obtained with current sequencing technologies. For instance, with the aim of exploring the diversity of small eukaryotes in the oceans all over the world, the expedition "Tara Ocean" [15] is generating more than 400 metagenomic samples containing each around 100 million short reads, that will need to be compared to each other.

Here, we introduce a time and memory-efficient method for extracting similar reads between two metagenomic datasets. The similarity is based on shared k-mers (words of length k). In order to fit with current memory capacities, the data structure we use is a modified version of a Bloom filter [16]. Bloom filters have recently been used in bioinformatics, notably for assembly graph partitioning [17], which enabled to perform metagenomic *de novo* assembly using 30x less memory.

This manuscript presents two main contributions: **(I)** a new algorithm, called Compareads, which computes the similarity measure between two metagenomics datasets; **(II)** a new simple but extremely efficient data structure based on the Bloom filter for storing the presence/absence of k-mers in huge datasets. The manuscript is organized as follows: in Section 1, we depict the Compareads algorithm and the new data structure. In Section 2 we provide results both about the data structure and about Compareads, showing the efficiency of our approach in term of computation time, memory and biological accuracy.

1 Methods

Preliminaries and definitions A *sequence* is composed by zero or more symbols from an alphabet Σ. In this work, as we are dealing with DNA, $\Sigma = \{A, C, G, T\}$. A sequence s of length n on Σ is denoted also by $s[0]s[1]\ldots s[n-1]$, where $s[i] \in \Sigma$ for $0 \leq i < n$. We denote by $s[i,j]$ the *substring* $s[i]s[i+1]\ldots s[j]$ of s. In this case, we say that the substring $s[i,j]$ occurs at position i in s. We call *k-mer* a sequence of length k, and $s[i, i+k-1]$ is a k-mer occurring at position i in s.

Overview of Compareads Compareads is designed for finding similar sequences between two read sets. This basic operation may appear extremely simple. However, it has to be highly efficient, in term of computation time and memory footprint, in order to scale with huge metagenomics datasets.

In order to perform efficiently this operation, Compareads indexes k-mers and uses a rough but efficient notion of "*similar sequences*" defined as follows:

Definition 1 (shared k-mer) *Two sequences s_1 and s_2 share a k-mer if and only if $\exists(i_1, i_2)$ such that $s_1[i_1, i_1 + k - 1] = s_2[i_2, i_2 + k - 1]$.*

Definition 2 (Similar sequences) *Given integers k and t, two sequences s_1 and s_2 are said similar if and only if they share at least t non overlapping k-mers.*

In a few words, given two read sets A and B, the goal of the Compareads algorithm is to find the subset of reads from A which are similar to a read in B such set being denoted by $(A \overrightarrow{\cap} B)$. As it is a heuristic (see Section 1.4), our algorithm outputs an over-approximation of set $(A \overrightarrow{\cap} B)$ denoted by $(A \overset{\rightsquigarrow}{\cap} B)$.

1.1 Computing $(A \overset{\rightsquigarrow}{\cap} B)$

Compareads computes $(A \overset{\rightsquigarrow}{\cap} B)$ in two steps. The **indexing** step consists in storing in memory all k-mers having at least one occurrence in the set B. The **query** step processes reads from set A one by one. For a read $r \in A$, the index is used to test for each k-mer of r if is present in the set B. If at least t non-overlapping k-mers are returned as present, then the read r is inserted in $(A \overset{\rightsquigarrow}{\cap} B)$. The main practical challenge faced by Compareads is to index the possibly huge volume of k-mers contained in B. The data structure must therefore fulfill three criteria: it must be quick to build, have a low memory footprint and be quick to request. Section 1.2.2 describes the chosen probabilistic data structure, based on a Bloom filter.

Limiting the indexation space To control the approximation error (see Section 1.4), the indexing phase is interrupted whenever the volume of k-mers in the first reads of B exceeds a fixed value n. The query phase is then performed on the whole A dataset. This phase returns a partial intersection between A and a first chunk of reads from B. The remaining partial intersections between A and the next chunks of reads from B (each representing a volume of n k-mers or less) are sequentially computed, until all the reads from B have been indexed. Eventually, Compareads returns the union of all partial intersections. Note that, in terms of results, this partitioning approach is strictly equivalent to performing a complete indexing of B then a query of all the reads from A. To avoid redundant computations, reads from A considered as "similar" in one of the partial intersections are tagged using a bitvector and are not queried further.

Time complexity Let n_A and n_B be the number of k-mers respectively in set A and set B. Computing $(A \overset{\rightsquigarrow}{\cap} B)$ is done in time $O(n_B)$ (indexation) + $O\left(n_A \times \frac{n_B}{n}\right)$ (query). The $\frac{n_B}{n}$ term is due to the limitation of the indexation space. In practice, for instance for a classical Illumina read set, this term is bellow 10.

1.2 Ad hoc data structure

The index data structure we use is based on a Bloom filter, specially designed for the task of storing efficiently a huge set of k-mers, while being fast to build and to query. We shortly recall in Section 1.2.1 what a Bloom filter is before describing, in Section 1.2.2, our data structure called BDS.

1.2.1 Bloom filter

A Bloom filter is a probabilistic data structure designed to test the membership of elements in a set [16]. It consists of an array of m bits, all initialized to zero, and a set of hash functions. Each hash function maps an element to a single position in the array. Each element is associated, through the values of the hash functions, to several positions in the array. To insert an element in the structure, the bits in the array associated to this element are all set to one. The structure answers membership queries by checking whether all the bits in the array associated to an element are set to one.

This data structure is probabilistic in nature, as false positives are possible. Even if an element is not in the set, its bits in the array may still be all set to one. This is because the bits associated to an element may independently be associated to other elements. Hence, the Bloom filter returns a wrong answer with non-zero probability. This probability is the *false positive rate*. An asymptotic approximation of the false positive rate is $0.6185^{m/n}$, assuming n elements are inserted in the m-bits array, and $(\ln 2 \cdot (m/n))$ hash functions are used [18]. False negatives never occur: if an element belongs to the set, the Bloom filter always answers positively. Bloom filters are space-efficient: only $(n \log_2 e \cdot \log_2(1/\epsilon))$ bits are required to support membership queries for n elements with a false positive rate of ϵ [18].

1.2.2 The Bloom Data Structure index

In this article, we consider a slightly different variation of Bloom filters: instead of using a single array of bits, each hash function corresponds to a distinct array, disjoint from all other functions. In terms of performance, with uniform hash functions, this variation is asymptotically equivalent to the original definition [18]. To avoid confusion with classical Bloom filters, we refer to this variation as BDS, standing for Bloom Data Structure.

Particular hash functions The hash functions used in this framework is a specific family of functions, which can be efficiently computed on consecutive k-mers. We consider the set of functions which map a k-mer to a bit sequence of length k, where each nucleotide is associated to a bit set to 0 or 1, depending only on its type (A, C, G or T). An exhaustive enumeration, in equations 1 and 2, shows that there exists only 7 functions in this set. We can distinguished two types, the first three, f_1, f_2 and f_3, are said to be *balanced* (equation 1), whereas the other four are said to be *unbalanced* (equation 2)

$$f_j : \Sigma^k \to \{0,1\}^k : \forall i \in [1,k] \begin{cases} f_1(s)[i] = 0 & \text{if} \quad s[i] = A \text{ or } C \quad f_1(s)[i] = 1 \quad \text{otherwise} \\ f_2(s)[i] = 0 & \text{if} \quad s[i] = A \text{ or } G \quad f_2(s)[i] = 1 \quad \text{otherwise} \\ f_3(s)[i] = 0 & \text{if} \quad s[i] = A \text{ or } T \quad f_3(s)[i] = 1 \quad \text{otherwise} \end{cases} \tag{1}$$

$$f_j : \Sigma^k \to \{0,1\}^k : \forall i \in [1,k] \begin{cases} f_4(s)[i] = 0 & \text{if} \quad s[i] = A \quad f_4(s)[i] = 1 \quad \text{otherwise} \\ f_5(s)[i] = 0 & \text{if} \quad s[i] = C \quad f_5(s)[i] = 1 \quad \text{otherwise} \\ f_6(s)[i] = 0 & \text{if} \quad s[i] = G \quad f_6(s)[i] = 1 \quad \text{otherwise} \\ f_7(s)[i] = 0 & \text{if} \quad s[i] = T \quad f_7(s)[i] = 1 \quad \text{otherwise} \end{cases} \tag{2}$$

One important property of these functions is that there is a simple relationship between the hash values of two consecutive k-mers in a read. One can see that the hash value of the next k-mer can be quickly computed, by left-shifting the binary sequence of the previous hash value and appending an extra bit. These functions are not classical hash functions, yet we show that they exhibit good hashing properties when applied to k-mers. In Section 2.1, the performance of these functions is compared with that of a classical hash function in terms of computation time, and false positive rate in the BDS.

1.3 The Compareads pipeline

Computing $(A \overset{\rightsquigarrow}{\cap} B)$ is asymmetrical. Indeed $(A \overset{\rightsquigarrow}{\cap} B)$ does not contain the reads from B which are similar to reads in A. For doing this, one needs to compute also $(B \overset{\rightsquigarrow}{\cap} A)$. In practice, for fully and symmetrically comparing two sets A and B we apply a pipeline slightly more complicated than simply $(A \overset{\rightsquigarrow}{\cap} B)$ followed by $(B \overset{\rightsquigarrow}{\cap} A)$. This whole pipeline, designed for reducing a heuristic effect is described in Section 1.4.1.

Similarity measure While comparing read sets A and B, the result provided by Compareads is composed of two sets: $(A \overset{\rightsquigarrow}{\cap} B)$ and $(B \overset{\rightsquigarrow}{\cap} A)$. Then, a similarity measure between the two datasets is computed as follows: $Sim(A,B) = \frac{\left|A \overset{\rightsquigarrow}{\cap} B\right| + \left|B \overset{\rightsquigarrow}{\cap} A\right|}{|A|+|B|} * 100$, where $|X|$ denotes the cardinality of the set X.

1.4 Dealing with false positives

Our approach may generate false positives for two reasons we describe in the two upcoming sections which also expose solutions for limiting these effects.

1.4.1 False positives due to k-mer shared between a read and a dataset

Using $t > 1$, Compareads algorithm can call similar sequences that do not respect strictly the definition of similarity given in definition 2. Indeed, steps described in Section 1.1 detect reads from A that share at least

t k-mers with reads from B. This is less stringent than finding reads from A that share at least t k-mers with at least one read from set B. In fact, the t k-mers found in read A are possibly spread over two or more distinct reads from set B.

This issue can be mitigated by performing the following steps to compute both $(A \overset{\rightsquigarrow}{\cap} B)$ and $(B \overset{\rightsquigarrow}{\cap} A)$:

1. Compute $(A \overset{\rightsquigarrow}{\cap} B)$, storing the results in a set denoted by $(A \overset{\rightsquigarrow}{\cap} B)^*$.
2. Compute $\left(B \overset{\rightsquigarrow}{\cap} (A \overset{\rightsquigarrow}{\cap} B)^*\right)$ storing the results in a set denoted by $(B \overset{\rightsquigarrow}{\cap} A)$.
3. Compute $\left(A \overset{\rightsquigarrow}{\cap} (B \overset{\rightsquigarrow}{\cap} A)\right)$ storing the results in a set denoted by $(A \overset{\rightsquigarrow}{\cap} B)$.

In a few words, the two output datasets $(B \overset{\rightsquigarrow}{\cap} A)$ and $(A \overset{\rightsquigarrow}{\cap} B)$ are obtained by applying the fundamental operation $(\overset{\rightsquigarrow}{\cap})$ between a query and a read set being itself already the result of the asymmetrical $(\overset{\rightsquigarrow}{\cap})$ operation. This enables to remove some false positives due to k-mers spread over several reads.

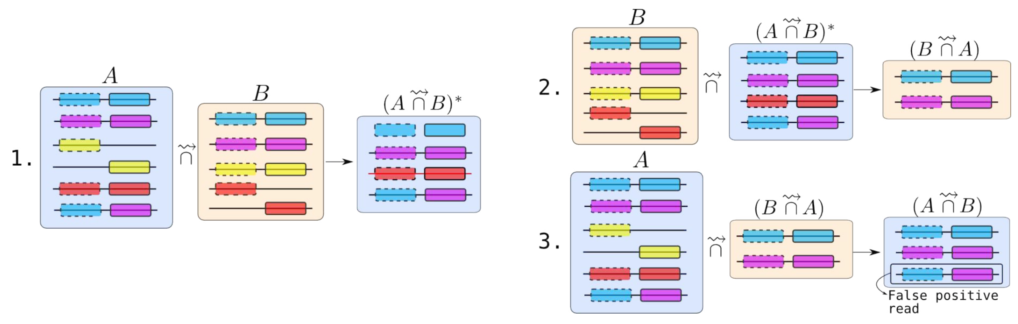

1.
A
$\overset{\rightsquigarrow}{\cap}$
B
$(A \overset{\rightsquigarrow}{\cap} B)^*$
2.
B
$\overset{\rightsquigarrow}{\cap}$
$(A \overset{\rightsquigarrow}{\cap} B)^*$
$(B \overset{\rightsquigarrow}{\cap} A)$
3.
A
$\overset{\rightsquigarrow}{\cap}$
$(B \overset{\rightsquigarrow}{\cap} A)$
$(A \overset{\rightsquigarrow}{\cap} B)$
False positive read

Figure 1: Representation of the three steps while comparing symmetrically read sets A and B. In each set, reads are represented by horizontal lines. On each read one or two shared k-mers are represented by rectangles.

The example presented in Figure 1 illustrates this issue for the case $t = 2$. The two first reads of sets A and B are similar. They are classically output by Compareads respectively in $(A \overset{\rightsquigarrow}{\cap} B)$ and $(B \overset{\rightsquigarrow}{\cap} A)$. The two next reads contain only one shared k-mer (yellow) with reads of set B, they are discarded. The next read of set A contains two (red) shared k-mers with two distinct reads in set B. After a first comparison, $(A \overset{\rightsquigarrow}{\cap} B)^*$ contains this false positive read. However, in step 2, while computing $(B \overset{\rightsquigarrow}{\cap} (A \overset{\rightsquigarrow}{\cap} B)^*)$, these two reads are not conserved in $(B \overset{\rightsquigarrow}{\cap} A)$. Thus, during step 3, the two red k-mers are not present anymore in set $(B \overset{\rightsquigarrow}{\cap} A)$ and thus are not present in $(A \overset{\rightsquigarrow}{\cap} (B \overset{\rightsquigarrow}{\cap} A))$. They are thus correctly absent from the final results $(A \overset{\rightsquigarrow}{\cap} B)$. However, the last read from set A is a case of false positive. It contains k-mers spread over distinct reads from B, the latter belonging to $(B \overset{\rightsquigarrow}{\cap} A)$. Thus, even during step 3, these two k-mers remain shared with reads from set $(B \overset{\rightsquigarrow}{\cap} A)$ and are output in $(A \overset{\rightsquigarrow}{\cap} B)$.

Note that in practice, the last set $(A \overset{\rightsquigarrow}{\cap} B)$ is obtained by computing $\Big((A \overset{\rightsquigarrow}{\cap} B)^* \overset{\rightsquigarrow}{\cap} (B \overset{\rightsquigarrow}{\cap} A)\Big)$ instead of simply $\Big(A \overset{\rightsquigarrow}{\cap} (B \overset{\rightsquigarrow}{\cap} A)\Big)$ (used here for simplifying the reading). This operation provides the same result but is computed faster as $|(A \overset{\rightsquigarrow}{\cap} B)^*| \leq |A|$.

As outlined in the example Figure 1, this pipeline may still conserve some false positives. The latter are characterized by the fact that they contain t shared k-mers with at least two distinct reads from the indexed dataset B themselves considered as similar to reads of set A. Even if this side effect is difficult to assess, we show in Section 2.2 that Compareads provides trustfull results, highly similar to a classical approach, on several real datasets.

1.4.2 Bloom filter false positives

As exposed in Section 1.2.2, the BDS index is a probabilistic data structure, that may consider a k-mer as indexed while this is not the case (*i.e.* a false positive or FP). Here, we analysed the variations of the false positive rate for each hash function and their combinations with respect to the parameter k and the number n of distinct indexed k-mers. This enabled then to choose the parameters and the appropriate combination of functions that give the best tradeoff between memory and false positive rate.

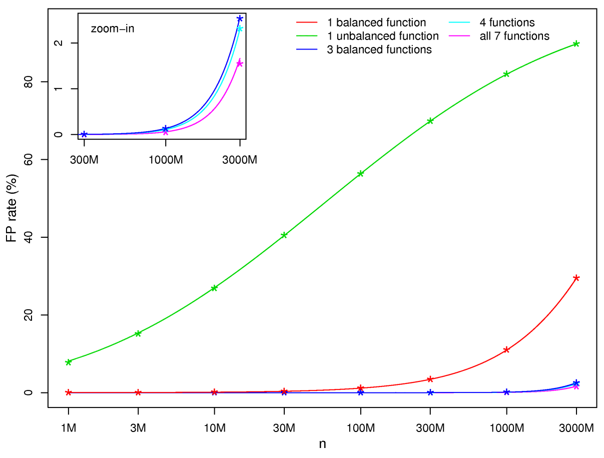
FP probablity for each function Assuming the nucleotide composition of the indexed k-mers and of the query k-mers are unbiaised, we can easily compute the probability, $P_{FP}(f_i, k, n)$, for any query k-mer to be a false positive with one of the seven hash functions, f_i (see the appendix Section A for details). The expressions of this probablity are presented in equations 3 for a balanced hash function and 4 for an unbalanced one.

$$\forall i \in \{1,2,3\} \quad P_{FP}(f_i, k, n) = 1 - (1 - \frac{1}{2^k})^n \tag{3}$$

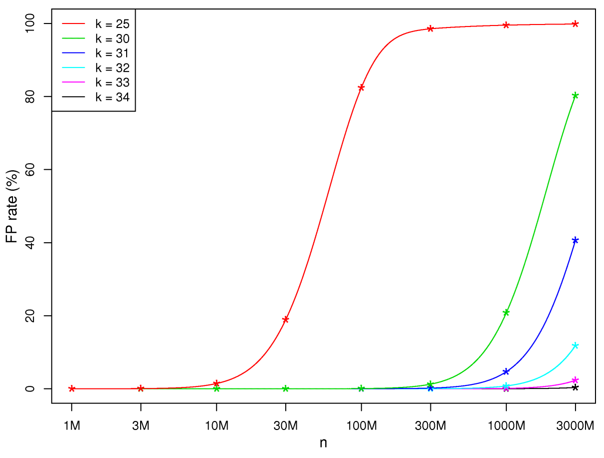
$$\forall i \in \{4,5,6,7\} \quad P_{FP}(f_i, k, n) = \sum_{x=0}^{k} \binom{k}{x} a_x (1 - (1 - a_x)^n) \qquad with \quad a_x = (\frac{1}{4})^x (\frac{3}{4})^{k-x} \tag{4}$$

We have plotted in Figure 2a the theoretical FP rate for both types of hash function, and we can see that balanced functions give much less false positives than unbalanced ones. This is due to the fact that balanced functions distributes the hash codes uniformally over the 2^k bit-array, while this not the case for the unbalanced ones.

FP probablity for a combination of functions One important property of the balanced hash functions is that there do not exist two distinct k-mers that have the same couple of hash codes with any two of these functions. This implies that the probability of having a FP with one function does not depend on the result



(a) FP rate for different combinations of hash functions



(b) FP rate for different values of k

Figure 2: FP rate as a function of the number of indexed k-mers (in log scale). Plain lines correspond to theoretical predictions, whereas star points correspond to empirical values obtained with simulations. **(a)** This figure was obtained for $k = 33$ for balanced and unbalanced functions and some combinations of them. The combination entitled “4 functions” is composed of the 3 balanced functions plus one unbalanced. **(b)** For several values of k, this figure was obtained for a combination of 4 hash functions: all three balanced plus one unbalanced.

with another function (or almost not, see details in appendix Section A). The probability of FP can then be easily computed as follows:

$$P_{FP}(f_1 \cap f_2 \cap f_3, k, n) \lesssim (1 - (1 - \frac{1}{2^k})^n)^3 \tag{5}$$

This “independence” property implies also that combining these 3 functions in our BDS is very efficient to reduce the FP rate, as can be seen in Figure 2a especially for large values of n

Concerning the unbalanced functions, this independence property is lost, since it is possible to find couples of distinct k-mers that share the same couple of hash codes for at least 2 of the unbalanced functions, or for one balanced function and at least one unbalanced. Therefore we could not figure out the theoretical FP rate. Nevertheless, we estimated it by simulations. We found that empirical results are very close to the formula obtained by multiplying the individual probabilities, *i.e.* assuming complete independence between all functions (Figure 2a). For details about how empirical results were obtained, see appendix Section B.

Choice of parameters The comparison of these FP rate curves led us to choose the combination of the three balanced functions plus an unbalanced one. This choice is motivated by the fact that unbalanced functions are not essential, as they have a limited effect on the FP rate (Figure 2a). Moreover, using these functions doubles the memory cost as storing four or less functions needs 2^{k-1} bytes, while seven functions need 2^k

bytes. Thus we choose to consume 2^{k-1} bytes memory, having space for four functions, naturally composed of the thee balanced plus one of the unbalanced functions.

For this combination of functions, we plotted the FP rate as a function of n and for several values of k in Figure 2b. The larger k is, the less FP we get for a given number of indexed k-mers. Consequently, for large values of k, more k-mers can be indexed while maintaining a reasonable FP rate. However, the memory allocated to BDS grows with k and larger values of k increases the stringency of our similarity measure. We can see in Figure 2b, that using k-mers of size at least 30 enables to index at least 300 millions of k-mers with less than 2% of false positives.

We used $k = 33$ and when indexing up to $n =$one billion distinct k-mers we obtain a theoretical upper bound of 0.13% of false positives (with 3 balanced functions, equation 5). The FP rate is even lower when adding one of the unbalanced function, we estimated it empirically to 0.114%.

2 Results

2.1 Practical performance of the BDS, comparison with other data structures

Since hash functions described in Section 1.2.2 have a fixed range, the memory used by the BDS depends only on the value of k and the number of hash functions used. Recall that each hash function is associated to a dedicated bit array which occupies 2^k bits. Using 7 hash functions, the BDS has a total memory footprint of $7 * 2^k$ bits and can be stored in 2^k bytes. When using only 4 hash functions, the BDS occupies $4 * 2^k$ bits and can be stored in 2^{k-1} bytes. As shown in Section 1.4.2, using 4 hash functions and $k = 33$ is a good set of parameters for indexing $n =$ one billion distinct k-mers. Using such parameters, the memory usage of Compareads is 4 GB.

We propose here a comparative analysis of the BDS with other data structures. In next Section we show that classical non probabilistic data structures result in a worst time and memory achievements, while in Section 2.1.2, we show that the BDS is the best suited for the problem of indexing huge amounts of k-mers.

2.1.1 Comparison with non probabilistic data structures: suffix array and hash table

Indexing n characters using the simplest version of a suffix array (not enhanced [19] and without LCP information) requires $5n$ bytes of memory [20]. Compared to our set of parameters where $n = 1$ billion, the memory footprint would be 5×10^9 bytes, *i.e.* 4.66 GB. While this is comparable to the BDS, the query time of the suffix array, $O(k \log n)$, is significantly worse.

An hash table can be used to store an exact set of k-mers. Such structure stores the k-mers explicitly, hence it requires at least $n \cdot \lceil \frac{2k}{8} \rceil \cdot 8$ bits (assuming no overhead), i.e. 16.5 GB for one billion of 33-mers.

2.1.2 Comparison with other hash functions and with a classical Bloom filter

Time comparison with other hash functions The hash functions defined for BDS were designed with speed in mind. In this paragraph, we compare them with a popular and fast hash function (`hashlittle2` from http://burtleburtle.net/bob/c/lookup3.c using Jenkins functions). We simulated 1 millon of 100-bp reads, where each nucleotide is drawn uniformly and independently. To simulate the behavior of computing hashes for the BDS, 4 hash values were computed for each 33-mer. For the `hashlittle2` function, we simulated this behavior by computing 4 hashes with 4 different initial values. We recorded the time required to compute the hashes for all the 33-mers present in the reads, averaged over 3 executions. Computing the hash with the `hashlittle2` function took 13.1 seconds (5.2 MHashes/s), whereas for the BDS hash functions, the same computation took 1.4 seconds (49.8 MHashes/s). Hence, the BDS hash functions are one order of magnitude faster than a classical set of hash functions.

FP rate comparison with other hash functions We can see in Figure 3 that the FP rate of classical hash functions follows the FP rate of our balanced functions (it follows the equation 5 with the exponent 3 being replaced by the number of functions used). However it diverges with more than 3 functions, as we could not add other balanced functions and we added in place unbalanced ones which have higher FP rates. Even if, for more than three functions, classical hash functions produce less FP, the difference with our BDS structure is small: for 1 billion indexed k-mers, combinations of 4 classical functions give 0.01% FP on average, compared to 0.114%. We chose to have a slithly higher FP rate, but with a significant gain in computing time.

Comparison with a classical Bloom filter A classical Bloom filter requires a fixed amount of memory to index n k-mers. Evaluating the Bloom filter memory using formula from Section 1.2.1 for one billion elements, with a false positive rate of 0.114%, yields 1.8 GB of memory. While this is twice smaller than the BDS, a classical Bloom filter would require classical hash functions. However, as shown above, classical hash functions are an order of magnitude slower to compute.

2.2 Comparing with a classical approach using BLAST

Our approach is an heuristic based on shared k-mers between reads. Here we compare Compareads with a well-established method, BLAST [21] that is based on sequence alignment. The dataset used is composed of 15 bacterial metagenomes obtained from fresh water with three different conditions of Carbon/Nitrogen ratio (unpublished data). On average, each sample is composed of 176409 reads with an average of 400

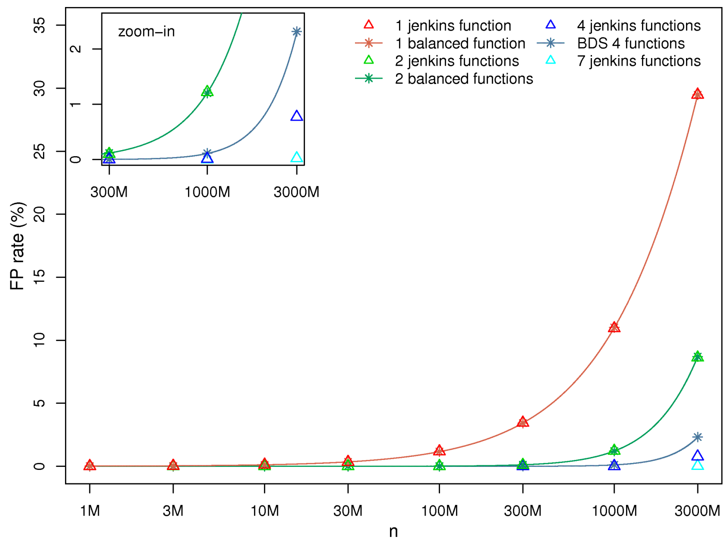


Figure 3: Comparison of FP rates between classical hash functions and the functions we used in the BDS. FP rate is plotted as a function of the number of indexed k-mers (in log scale), with $k = 33$. Plain lines correspond to theoretical predictions for the balanced functions (BDS), whereas star points and triangles correspond to empirical values obtained with simulations using respectively BDS functions and classical hash functions. The combination entitled "BDS 4 functions" is the one chosen for Compareads and is composed of the 3 balanced functions plus one unbalanced.

nucleotides per read (Roche 454 technology).

Both Blast and Compareads were used to compute all of the 120 pairwise intersections between the 15 datasets. Blast was configured to find similar sequences between two samples with a local alignment greater than 80 nucleotides and more than 90% of sequence identity. Compareads was used to find sequences sharing respectively t =1, 4 and 10 k-mers of 33 nucleotides. As shown in Table 1, computing one intersection

	Total Time (min)	Mean Time for one intersection (s)	Reads Found
Blast	7200	3600	33 400 091
Compareads $1 * 33$	238	119	35 898 023
Compareads $4 * 33$	230	115	31 997 243
Compareads $10 * 33$	228	114	21 350 268

Table 1: The CPU time per intersection and the global CPU time using a single core of an Intel® Xeon® CPU X5550 at 2.67GHz. **Reads Found** corresponds to the total number of similar reads in all the 120 intersections.

between two samples using Compareads is more than 30 times faster than using Blast for a close total number of similar reads.

For each experimentation, samples were hierarchically clustered based on their pairwise similarity scores and then drawn as a dendrogram. As shown in Figure 4, the dendrogram obtained with the Blast approach (a) is slightly different but the three main branches are the same than with the Compareads approach (b). Interestingly, these branches discriminate three groups of samples corresponding to the three different biological conditions indicated by 1, 10 and 40 in the samples names: 1 corresponds to addition of Carbon in the water, 10 stands for normal condition and 40 for introduction of Nitrogen. Notably, all dendrograms

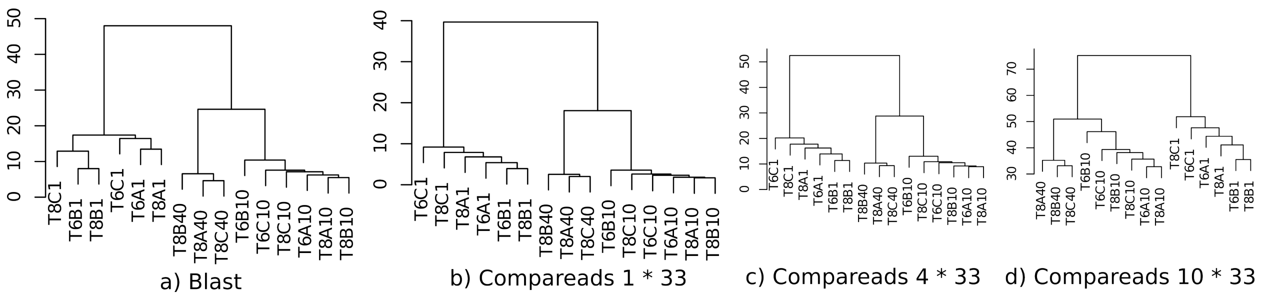


Figure 4: Representation of hierarchical clustering based on pairwise intersections between all samples using Blast (a) and Compareads (b, c, d).

based on Compareads approach (b, c, d) show a similar organization. Increasing the number of shared k-mers leads to be more stringent and decreases the number of similar reads but do not affect the global organization of the dendrogram, demonstrating the robustness of our similarity measure.

2.3 Applying Compareads to Global Ocean metagenomic samples

We tested Compareads on a larger and famous public dataset from the Global Ocean Sampling (The Sorcerer II expedition) [22]. It is composed of 44 samples from the microbial world of seawater, collected across several thousand of kilometers from the Northwest Atlantic through the Eastern Tropical Pacific oceans and for which an analysis of similarity between samples has been done [22]. The whole dataset is composed of 44 samples containing each on average 174759 long reads (1249 nucleotides per read on average, Sanger technology). Compareads compute all of the 990 intersections in 72 hours and half: on average, one intersection was performed in 4 minutes and 23 seconds on a single core of an Intel® Xeon® CPU X5550 at 2.67GHz. Results presented in Figure 5 are highly similar to those presented in the original publication [22], p.418. Two main groups are well discriminated. The first one, represented in turquoise-blue, groups together almost all samples coming from temperate seawater of the North American East Coast except the 14 one, like in the original study. This group also contains two samples really different from all others: the first contains freshwater and the second hypersaline water. The dark-green part corresponds to samples coming from the north part while light-green one gathers samples from the south part. Orange samples correspond to estuary. All of those three groups are identical to the original study. The second main part, colored in yellow, groups together datasets of tropical and Sargasso seawater. The dark-blue part aggregates samples coming only from Galapagos Islands. Red square delimitates Sargasso Sea samples. On the original study, the sample 00a is not in this group. According to metadata, the gray part, like in the original publication, is composed of various samples. Finally, purple samples regroup both Caribbean Sea and some Open Ocean datasets, as

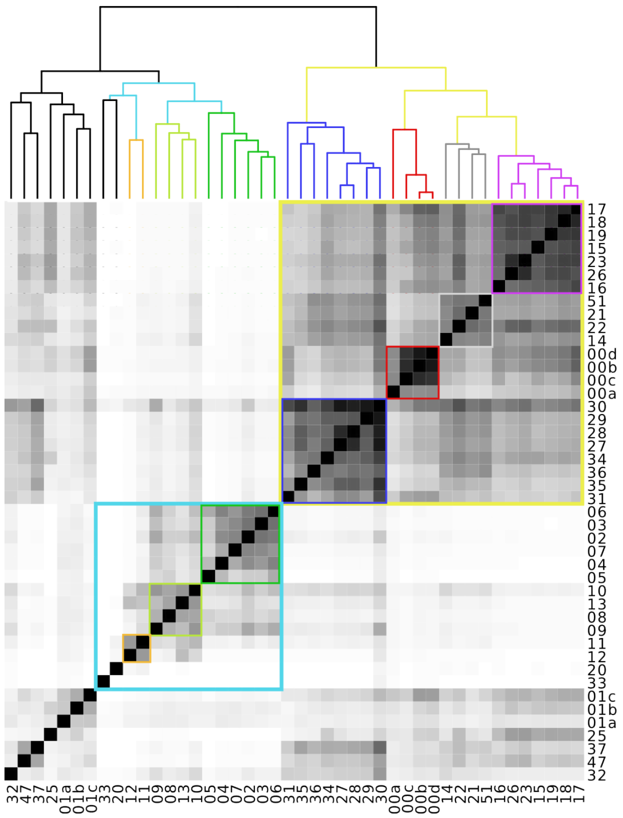


Figure 5: Similarity matrix resulting from the comparison of 44 samples from The *Sorcerer II* Global Ocean Sampling Expedition using Compareads. Grey levels correspond to similarity levels, intersections with more than 50% of similarity are in black. The two main groups, in turquoise-blue and yellow, correspond respectively to north American east coast and tropical samples.

the original study.

Those results show that Compareads can also be used on Sanger reads and deliver reliable biological conclusions. Indeed, despite of false positives and the simple definition of similarity, we were able to retrieve the classification of metagenomes according to their geographical origin.

Conclusion

Motivated by *de novo* comparative metagenomics, this paper proposes two main contributions. The first one is a data structure based on Bloom filters that can index, for instance up to one billion distinct words of length 33 (33-mers) using 4Gb of memory, with an error rate of 0.11%, and that is faster to build and request, to the best of our knowledge, that any other existing data structure. The second main contribution is a software, called Compareads which uses this data structure to efficiently perform *de novo* intensive comparisons of huge metagenomic datasets generated by High Throughput Sequencers. We have shown that this approach enables to retrieve and classify differences in species content between metagenomic samples. For this kind of comparison, our approach is much faster than alternative ones such as Blast and thus enables to scale to huge datasets. For instance, we tested the scalability of Compareads on a large oceanic unpublished dataset, from the Tara Ocean expedition [15]; it is composed of 31 metagenomes and contains overall 3.5 billions of Illumina short reads (108bp). Each intersection was performed in 10 hours and 55 minutes in average using 4Gb of memory. Such features enabled us to compute the $\frac{31*32}{2} = 496$ metagenome datasets intersections in 6 days and 10 hours using 50 cores of Intel® Xeon® CPU X5550 at 2.67GHz. This would have been unfeasible with any other known existing tools (based on results 2.2, Blast is about 30 times longer and would take more than 6 months to complete this task with the same resources).

Compareads has been conceived for being parallelizable both at fine and coarse grained levels. Future work will consist in implementing a Compareads parallel version exploiting multi-core and GPU chips.

Compareads is released under the CeCILL license and can be freely downloaded from http://alcovna.genouest.org/compareads/.

Author's contributions

DL and PP initiated the work. RC and CL provided expertise about Bloom filters datastructures and their statistical aspects. NM and PP made the implementations. NM, CL, DL and PP performed the experiments. All authors participated to the redaction and approved the final manuscript.

Acknowledgements

This work was supported by the french ANR-2010-COSI-004 MAPPI Project. Authors warmly thank O. Jaillon from the Genoscope and P. Vandenkoornhuyse from the Ecobio UMR for providing their biological expertise and metagenomic datasets. Additionally, we thank G. Rizk for its help and comments with the data structure and F. Gauthier for the "$\overset{\leadsto}{\cap}$" symbol creation.

References

1. Desai N, Antonopoulos D, Gilbert JA, Glass EM, Meyer F: **From genomics to metagenomics**. *Curr. Opin. Biotechnol.* 2012, **23**:72–76.
2. Johnson DS, Mortazavi A, Myers RM, Wold B: **Genome-wide mapping of in vivo protein-DNA interactions.** *Science (New York, N.Y.)* 2007, **316**(5830):1497–502.
3. Licatalosi DD, Mele A, Fak JJ, Ule J, Kayikci M, Chi SW, Clark TA, Schweitzer AC, Blume JE, Wang X, Darnell JC, Darnell RB: **HITS-CLIP yields genome-wide insights into brain alternative RNA processing**. *Nature* 2008, **456**(7221):464–469.
4. Davey JW, Blaxter ML: **RADSeq: next-generation population genetics**. *Briefings in Functional Genomics* 2010, **9**(5-6):416–423.
5. Wooley JC, Godzik A, Friedberg I: **A Primer on Metagenomics**. *PLoS Comput Biol* 2010, **6**(2):e1000667.
6. Amann RI, Ludwig W, Schleifer KH: **Phylogenetic identification and in situ detection of individual microbial cells without cultivation**. *Microbiol. Rev.* 1995, **59**:143–169.

7. Mende DR, Waller AS, Sunagawa S, Jäelin AI, Chan MM, Arumugam M, Raes J, Bork P: **Assessment of Metagenomic Assembly Using Simulated Next Generation Sequencing Data**. *PLoS ONE* 2012, **7**(2):e31386.

8. Wang Y, Leung HC, Yiu SM, Chin FY: **MetaCluster 4.0: a novel binning algorithm for NGS reads and huge number of species**. *J. Comput. Biol.* 2012, **19**(2):241–249.

9. Markowitz VM, Chen IM, Chu K, Szeto E, Palaniappan K, Grechkin Y, Ratner A, Jacob B, Pati A, Huntemann M, Liolios K, Pagani I, Anderson I, Mavromatis K, Ivanova NN, Kyrpides NC: **IMG/M: the integrated metagenome data management and comparative analysis system**. *Nucleic Acids Res.* 2011.

10. Huson DH, Auch AF, Qi J, Schuster SC: **MEGAN analysis of metagenomic data**. *Genome Res.* 2007, **17**(3):377–386.

11. Foerstner KU, von Mering C, Hooper SD, Bork P: **Environments shape the nucleotide composition of genomes**. *EMBO Rep.* 2005, **6**(12):1208–1213.

12. Raes J, Korbel JO, Lercher MJ, von Mering C, Bork P: **Prediction of effective genome size in metagenomic samples**. *Genome Biol.* 2007, **8**:R10.

13. Jaenicke S, Ander C, Bekel T, Bisdorf R, Droge M, Gartemann KH, Junemann S, Kaiser O, Krause L, Tille F, Zakrzewski M, Puhler A, Schluter A, Goesmann A: **Comparative and joint analysis of two metagenomic datasets from a biogas fermenter obtained by 454-pyrosequencing**. *PLoS ONE* 2011, **6**:e14519.

14. Sommer MO, Dantas G, Church GM: **Functional characterization of the antibiotic resistance reservoir in the human microflora**. *Science* 2009, **325**(5944):1128–1131.

15. Karsenti E: **Towards an 'Oceans Systems Biology'**. *Molecular Systems Biology* 2012, **8**(575):1–2.

16. Bloom BH: **Space/time trade-offs in hash coding with allowable errors**. *Commun. ACM* 1970, **13**(7):422–426.

17. Pell J, Hintze A, Brown T, Canino-koning R, Howe A, Tiedje JM: **Scaling metagenome sequence assembly with probabilistic de Bruijn graphs**. *PNAS.* in press.

18. Broder A, Mitzenmacher M: **Network applications of bloom filters: A survey**. *Internet Mathematics* 2004, **1**(4):485–509.

19. Abouelhoda MI, Kurtz S, Ohlebusch E: **Replacing suffix trees with enhanced suffix arrays**. *Journal of Discrete Algorithms* 2004, **2**:53–86.

20. Vyverman M, De Baets B, Fack V, Dawyndt P: **Prospects and limitations of full-text index structures in genome analysis.** *Nucleic acids research* 2012, :1–23.

21. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ: **Basic local alignment search tool**. *J. Mol. Biol.* 1990, **215**:403–410.

22. Rusch DB, Halpern AL, Sutton G, Heidelberg KB, Williamson S, Yooseph S, Wu D, Eisen JA, Hoffman JM, Remington K, Beeson K, Tran B, Smith H, Baden-Tillson H, Stewart C, Thorpe J, Freeman J, Andrews-Pfannkoch C, Venter JE, Li K, Kravitz S, Heidelberg JF, Utterback T, Rogers YH, Falcon LI, Souza V, Bonilla-Rosso G, Eguiarte LE, Karl DM, et al: **The Sorcerer II Global Ocean Sampling expedition: northwest Atlantic through eastern tropical Pacific**. *PLoS Biol.* 2007, **5**(3):e77.

Appendix

A Theoretical details for the false positive rate

As exposed in Section 1.2.2, the BDS index is a probabilistic data structure, that may consider a k-mer as indexed while this is not the case (*i.e.* a false positive). Here, we tried to express the false positive rate for each hash function that we defined in Section 1.2.2 and their combinations with respect to the parameter k and the number n of distinct indexed k-mers.

False positive probablity for each function Assuming the base composition of the indexed and query k-mers is unbiaised, we can easily compute the probability, $P_{FP}(f, k, n)$, for any query k-mer to be a false positive with one of the seven hash functions, f. This probability depends on the number of distinct k-mers sharing the same hash code. We can notice that for the balanced functions, f_1, f_2 and f_3, each 0 and 1 value can come from exactly 2 distinct nucleotides, thus the number of k-mers sharing the same hash code is the same for all k-mers and equals: 2^k. The probability for 2 k-mers to have distinct hash codes is then $1 - \frac{2^k}{4^k} = 1 - \frac{1}{2^k}$, and therefore the probability to have at least one k-mer among the n that are indexed sharing the same hash code is:

$$\forall i \in \{1,2,3\} P_{FP}(f_i, k, n) = 1 - (1 - \frac{1}{2^k})^n \qquad ((3))$$

Note that this corresponds to the false positive probability of any hash function that distributes the hash codes uniformly in a 2^k bit-array, such as those inspired of Jenkins functions, used as a comparison in Section 2.1.2.

As for the unbalanced functions, since the 0 bit-value encodes only one base, the number of k-mers sharing the same hash code depends on the number of 0 in the hash code of the query. For a given query k-mer with a hash code having x 0 the above probability for functions f_4, f_5, f_6 and f_7 becomes: $1-(1-(\frac{1}{4})^x(\frac{3}{4})^{(k-x)})^n$. To obtain the probability for any kmer, we have to sum over the different values of x the latter probability weigthed by the probablity for a k-mer hash code to have x 0. The composition of a given base in a k-mer of length k, assuming unbiased nucleotide composition, follows a binomial distribution, thus we get:

$$\forall i \in \{4,5,6,7\} \quad P_{FP}(f_i, k, n) = \sum_{x=0}^{k} \binom{k}{x} a_x (1 - (1 - a_x)^n) \qquad with \quad a_x = (\frac{1}{4})^x (\frac{3}{4})^{k-x} \qquad ((4))$$

$\binom{k}{x}$ being the binomial coefficient, ie $\binom{k}{x} = \frac{k!}{x!(k-x)!}$.

We can see in Figure 2 that balanced functions give much less false positives than unbalanced ones. This can be explained by the fact that for unbalanced functions, for a given k-mer with a "normal" composition and thus 25% of 0 in its hash-code, there are many more k-mers with the same hash-code than for a balanced function: $3^{\frac{3k}{4}} \gg 2^k$.

FP probablity for a combination of functions When combining several functions in our BDS, in order to have a false positive for a query k-mer all functions must return a false positive. As concerns the balanced function, we can easily see that for a given kmer, we can not find another k-mer that is a false positive simultaneously for any two 2 of these functions. In other words, there do not exist two distinct k-mers that have the same couple of hash codes with any two of these functions. This implies that the probability of having a false positive with one function does not depend on the result with another function, apart from the fact that the effective number of indexed k-mers that can be a false positive (n in equation 3) is reduced: indeed if x k-mers have the same hash code for one function, these k-mers have a null probability of having the same hash code for another function. Note that this effect can be neglicted given that n is very large. Therefore the product of individual probabilities for each balanced function gives the following upper bound:

$$P_{FP}(f_1 \cap f_2 \cap f_3, k, n) \lesssim (1 - (1 - \frac{1}{2^k})^n)^3 \qquad ((5))$$

Concerning the unbalanced functions, this independence property is lost, since it is possible to find a single k-mer that is a false positive for at least 2 of the unbalanced functions, or for one balanced function and at least one unbalanced. Therefore we could not figure out the theoretical false positive rate, or even an upper bound.

B Empirical estimation of false positive rate

For estimating the false positive rate of the BDS filled up with n k-mers, we made the following experiment. We generated n+100000 distinct random k-mers, used first for filling up the BDS with n k-mers and then for querying the BDS with the 100000 remaining k-mers. All k-mers being distinct, if the BDS answers "yes" while querying the presence of a k-mer, it is a false positive.

The generation of a huge set of x distinct random k-mers (with $x < 4^k$) is not trivial. This was done by dividing the space of 4^k k-mers into x non overlapping blocks, and them by picking up a random k-mer into each block. This method enables to uniformly cover the whole k-mer space.