`fimpera`: drastic improvement of Approximate Membership Query data-structures with counts

Lucas Robidou and Pierre Peterlongo

Univ. Rennes, Inria, CNRS, IRISA, Rennes, France
{lucas.robidou,pierre.peterlongo}@inria.fr

Abstract.
Motivations: Approximate membership query data structures (AMQ) such as Cuckoo filters or Bloom filters are widely used for representing and indexing large sets of elements. AMQ can be generalized for additionally counting indexed elements, they are then called "counting AMQ". This is for instance the case of the "counting Bloom filters". However, counting AMQs suffer from false positive and overestimated calls.
Results: In this work we propose a novel computation method, called `fimpera`, consisting of a simple strategy for reducing the false-positive rate of any AMQ indexing all k-mers (words of length k) from a set of sequences, along with their abundance information.
This method decreases the false-positive rate of a counting Bloom filter by an order of magnitude while reducing the number of overestimated calls, as well as lowering the average difference between the overestimated calls and the ground truth. In addition, it slightly decreases the query run time. `fimpera` does not require any modification of the original counting Bloom filter, it does not generate false-negative calls, and it causes no memory overhead. The unique drawback is that `fimpera` yields a new kind of false positives and overestimated calls. However their amount is negligible. `fimpera` requires a unique parameter, and its results are only little impacted when using this parameter within recommended values.
As a side note, for the algorithmic needs of the method, we also propose a novel generic algorithm for finding minimal values of a sliding window over a vector of x integers in $O(x)$ time with zero memory allocation.
Availability: `https://github.com/lrobidou/fimpera`
Keywords: data structure; indexation; k-mers; counting Bloom filters; sequence data; abundance; AMQ

1 Introduction

Public data banks providing sequencing data or assembled genome sequences are growing at an exponential rate [4], faster than computational power. Searching a sequence of interest among datasets is a fundamental need. For instance it enables to better understand genetic changes in the tumour, offering precious information about the diagnosis and treatment of cancer [14], or it enables to study at a large scale the distribution and adaptation of life in oceans [13]). However no

method scales to the dozens of petabytes of data already available today. Thus, new computational methods are required to perform a search against datasets.

Querying datasets can be done precisely by aligning genome sequences (e.g. using Blast-like [1] algorithms), however aligning sequences is computational-resources intensive. Thus queries on large scale datasets are usually done through k-mers presence / absence. Basically, datasets are represented as their set of k-mers and queries are represented as their sequence of k-mers.

Methodological developments have thus been made to index every k-mers of a dataset. Some methods use Approximate Membership Query data structures (AMQ), e.g. bloom filters, to store presence / absence of k-mers, as SBT [12] or HowDe-SBT [5]; see [9] for a survey of approaches to index large dataset. However, very few methods tackle the issue of recording the abundance of the indexed k-mers. The abundance information is however crucial for many biological applications such as transcriptomics or metagenomics. Storing abundance is costly with regard to space consumption. Conversely, adding abundance information in an AMQ, turning it into a `counting AMQ`, without allocating more space increases drastically its false positive rate. As an example, BIGSI [3] relies on Bloom filters with a high false-positive rate, e.g. 25% false-positive rate per k-mer query. At constant memory usage, adding the abundance information would yield an extremely high false-positive rate. As such, methods storing abundances mostly rely on compression by clustering abundance with neighbouring k-mers or across datasets, as Reindeer [10] or Counting de Bruijn graphs [6]. These methods do not rely on `counting AMQ`, but rather on exact data structures.

In this paper, we do not propose a novel `counting AMQ`, but rather a wrapper to improve any existing `counting AMQ`, like a `counting Bloom Filter`. The method we introduce is called `fimpera`. It generalises one of our previous contribution [11]. In short, `fimpera` splits every k-mer into s-mers (with $k \geq s > 0$) and then associates the abundance of a k-mer to its constituent s-mers in a `counting AMQ`. This allows us to retrieve the abundance of a k-mer at query time via its s-mers count. Compared to the original `counting AMQ` indexing k-mers, we show that `fimpera` improves the abundance correctness while reducing the false positives rate by an order of magnitude without generating false-negative calls nor underestimation of the abundance of a k-mer.

Additionally, the `fimpera` algorithmic needs led us to propose a novel algorithm for computing in $O(x)$ time and with no memory allocation the sliding window minimums (resp. maximums). These are the minimal (resp. maximums) values of all sub-arrays of a fixed size over an array of x values. This contribution may be useful independently from the `fimpera` context. Its novelty is that, while being destructive for the input array, it uses no additional memory while other approaches use memory linear with the size of the intervals.

2 Methods

2.1 Background

Preliminary definitions

A k-mer is a word of length k over an alphabet Σ. Given a sequence S, $|S|$ denotes the length of S.

In the current framework, we consider a dataset as composed of one sequence or a multisets of sequences. Given a dataset D, $\mathcal{D}_k$ denotes multiset of k-mers extracted from D.

We denote the abundance of a k-mer d (the number of time d appears) in $\mathcal{D}_k$ by $abundance(\mathcal{D}_k, d)$. We consider that a k-mer is "present" (resp. "absent") in $\mathcal{D}$ if $abundance(\mathcal{D}_k, d) > 0$ (resp. $abundance(\mathcal{D}_k, d) = 0$).

A `counting AMQ` data structure represents a multiset of elements $\mathcal{D}_k$. It can be queried with any element d; the query's response on an `counting AMQ`, noted n, is always either correct or overestimated, i.e. $n \geq abundance(\mathcal{D}_k, d)$. If $n = abundance(\mathcal{D}_k, d)$, the `counting AMQ` reported the correct abundance, otherwise it reported an overestimation. Note that underestimation is not possible.

In particular, if $abundance(\mathcal{D}_k, d) = 0$ and $n > 0$, then d is found in the `counting AMQ` even if it is absent from $\mathcal{D}$. This particular case is a false positive call. The false-positive rate of a `counting AMQ`, denoted by FPR_{cAMQ}, is defined by $FPR_{cAMQ} = \frac{\#FP}{\#FP+\#TN}$ with $\#FP$ and $\#TN$ denoting respectively the number of false-positive calls and the number of true negative calls ($n = 0$). FPR_{cAMQ} depends on the used `counting AMQ` strategy and on the amount of space used by this `counting AMQ`.

2.2 Overview of `fimpera`

In this work, we focus on decreasing the false positive rate of a `counting Bloom Filter` (`cBF` for short). However, `fimpera` is a generic approach that may be applied on any `counting AMQ`. A `counting Bloom Filter` is a generalization of Bloom filters: each element is inserted along with its abundance instead of its presence only. This requires a few bits per entry for storing this information. Hence, either at constant size, a `counting Bloom Filter` has a higher false-positive rate than a simple Bloom filter, or are `counting Bloom Filter` requires more memory than a Bloom filter to achieve the same false-positive rate.

`fimpera`'s objectives are to reduce FPR_{cAMQ} and to improve precision on true positive calls, using a method based on splitting k-mers into smaller words called s-mers.

Indexation overview At indexation time, `fimpera` takes a file of counted k-mers, typically extracted from a genomic sequence dataset, and splits each k-mer to be indexed into its $k - s + 1$ s-mers ($k \geq s$). Each s-mer is then stored in a `cAMQ` along with its $s_{abundance}$. The $s_{abundance}$ of a s-mer is the maximum of the abundance of the k-mers containing this s-mer. We explain this choice in the following.

In the following, we set $z = k - s$, hence $z \geq 0$.

Query overview The query of `fimpera` consists of a set of sequences. For each sequence S, `fimpera` extract s-mers, which are then queried against the `cAMQ`, and the abundance of any k-mer of S is computed as the minimum of $s_{abundance}$ of its s-mers. By default, `fimpera` prints each input sequence along with the abundance of every of its consecutive k-mer. In the biological context, the input file is a fasta/fastq file (or a gzipped fasta/fastq file) containing reads.

`fimpera` is built in a modular way. Changing the default output (e.g. storing results instead of printing, computing average abundance per sequence or printing only sequences whose average k-mer abundances is above a user-defined threshold) can be programmed via inheritance of an Abstract Class.

Overview of false positive calls of `fimpera`'s query Let's consider a k-mer d with an abundance of 0 and each of its s-mer has an $s_{abundance}$ of 0 as well. With `fimpera`, wrongly reporting d as present requires that *every* s-mer of that k-mer are wrongly found as present in the `counting AMQ`. The probability of such an event is roughly FPR_{cAMQ}^{z+1}, leading to a dramatic decrease in the occurrences of false-positive calls with respect to z. For instance, with z=3 (which is a recommended and default value), a `counting AMQ` having a false positive rate of 25%, the probability of false-positive rate with `fimpera` for that setting is $\approx 0.04\%$.

The `fimpera` approach may generate a novel kind of false-positives. A queried k-mer, absent from the indexed dataset, may be composed of s-mers, all existing in this indexed set. Querying such k-mer with `fimpera` returns a non-zero abundance, so generating a false-positive, that we call a "construction false-positive". This new kind of false-positive call is specific to the `fimpera` approach.

Overview of overestimations of `fimpera`'s query To overestimate the abundance of a queried k-mer, overestimations are required to happen on the abundance of **every** s-mer of that k-mer. The more s-mer per k-mer, the more s-mer abundance overestimations need to happen to overestimate a k-mer abundance. s-mer abundance overestimations come from two sources:

- a collision occurs in the counting Bloom filter, leading to the overestimation of the less abundant colliding s-mer; and/or:
- a s-mer is shared among two different k-mers having different abundances. This overestimates the abundance of this s-mer of the least abundant k-mer. This happens not matter the false positive rate of the counting Bloom filter. We call those overestimations "construction overestimation"; this new kind of overestimation is specific to `fimpera`.

Observe a case of interest: consider two k-mers a and b overlapping over $k-1$ characters. If b has an abundance greater than a, then the correct abundance of a is retrievable through a unique s-mer (the unique s-mer of a that does not appear in the k-mer b). In such case, a is likely to be overestimated.

Consequently, `fimpera`'s overestimations are not uniformly distributed random events. Overestimations tend to be close to a change in abundance along

queried sequences. Furthermore, overestimations tend to raise k-mer's abundance close to the abundance of their neighbor k-mers, mitigating the impact of those overestimated calls. In the results, we show that the erroneous abundance calls are closer to the ground truth with `fimpera` compared to those obtained with the original `cBF`.

We now describe in more details both indexing step and querying step of `fimpera`, as well as two optimisations, allowing querying in constant time and skipping unnecessary queries.

2.3 Indexing with `fimpera`

As stated in Section 2.2, `fimpera`'s indexation is a two-step process:

- k-mers' abundances are computed from the input dataset (e.g. using KMC [7]);
- s-mers from these k-mers are stored in a `counting AMQ` together with their $s_{abundance}$. The $s_{abundance}$ of a s-mer is formally defined as the maximal abundance of the indexed k-mers in which this s-mer occurs.

Note that the $s_{abundance}$ of a s-mer α is lower or equal to the abundance of α in the input dataset. For instance consider a s-mer α that occurs in two k-mers respectively with an abundance of one and two. Then, the abundance of α is three ($= 1 + 2$), while the $s_{abundance}$ of α is two ($= \max(1, 2)$). Storing the $s_{abundance}$ of α instead of its abundance, enables to lower the abundance overestimations, as it avoids to accumulate the abundances of distinct k-mers it belongs to.

2.4 Querying with `fimpera`

`fimpera`'s query consists in querying all consecutive, overlapping k-mers from a sequence of size greater than k through their constituent s-mers. `fimpera`'s query is a two-step process:

- for every position in the query except the last $s - 1$ ones, s-mers starting at these positions are queried in the `counting AMQ` and stored in a array of integers $s_{abundances}$;
- The abundance any k-mer starting position p is the minimum value of the sub-array of length $(z + 1)$ starting at the position p: $s_{abundances}[p; p + z]$.

This non-optimised algorithm version of `fimpera`'s query is shown in algorithm 1, in supplementary material, Section S1.1.

This approach can be improved in two ways, in one way avoiding to recompute the minimal value of an array of length $z + 1$ for each position p, and in an other way by skipping some unnecessary s-mer queries. The two following Sections present these two optimisations.

2.5 Optimisations

Optimisation 1: sliding window minimums algorithm

The problem, independent of `fimpera`, is as follows: given a vector of values (integers of floats) v and an integer $size_window$, give an array r such that $\forall i \in [0, |v| - size_window], r[i] = min(v[i], v[i+1], ..., v[i + size_window - 1])$. The naive approach is to compute every window and search for the minimum in those windows. This algorithm is in $O(size_window \times |v|)$ time. We propose a solution in $O(|v|)$ time. Note that this is a classical problem for which non-published solutions can be found. However, the novelty of our proposed solution is that it does not require allocating any memory from the heap (which is slow for most systems).

The main idea is to split the input vector of values in fixed, non-overlapping windows of size $size_window$. Then, for each so called "fixed window", compute two vectors:

- min_left_j: $min_left_j[i]$ contains the minimum value encountered in the j-th fixed window up to the position i
- min_right_j: $min_right_j[i]$ contains the minimum value from the position i up to the end of the j-th fixed window

All min_left_j and min_right_j vectors are then concatenated into two vectors (min_left and min_right). The minimum of a *sliding* window starting at position i is thereupon the minimum between:

- $min_left[i + size_window - 1]$ (the minimum of the left part of the next fixed window)
- $min_right[i]$ (the minimum of the right part of the current fixed window)

An example is provided in Table 1.

i	0	1	2	3	4	5	6	7	8	9
v	5	3	7	1	4	5	3	2	2	3
j	0			1			2			3
min_left	5	3	3	$\underline{1}$	1	1	3	2	2	3
min_right	3	$\underline{3}$	7	$\underline{1}$	4	5	2	2	8	3
$min_sliding$	3	$\underline{\mathbf{1}}$	1	1	3	2	2	2		

Table 1. Computation example of the $min_sliding$ vector, with a window of size 3. Tables min_left and min_right are represented for helping the comprehension, but are not implicitly created in practice. The j row indicates the starting positions of the fixed windows. As a example, the minimal value of the sliding window of size 3 starting position $i = 1$ is $min_sliding[1] = 1$ (bold underlined value), being equal to $\min(min_left[1 + 3 - 1], min_right[1]) = \min(1, 3)$ (underlined values).

Note that, as described previously, this approach would require allocating memory for two vectors per call. This memory need may appear negligible in

theory as those vectors are limited by the query size which is a few hundred to few thousands. However, in practice, allocating memory for these vectors is time consuming, and may increase significantly the practical running time. We overcame this memory need thanks to these three following tricks. **1/** we compute $min_left[i]$ on the fly ($min_left[i] = min(min_left[i-1], v[i])$). **2/** min_right is computed *directly in the queried vector*. This does not impact the correctness of the algorithm, as $min_sliding \leq min_right[i] \leq v[i]$. **3/** the response (minimal value per sliding window) can be stored directly in the input queried vector as well. At the price of modifying the input vector, this allows the algorithm to be run in $O(size_query)$ time and to avoid any time consuming heap allocation.

A complete description of the optimised solution is provided in supplementary materials, Section S1.2, algorithm 2.

This algorithm offers a generic solution for computing the minimal value of a sliding window in constant memory and linear time. Its usefulness is not limited to `fimpera`. Note also that it can be straightforwardly modified for computing the maximal value instead of the minimal value of each window.

Optimisation 2: skip unnecessary *s*-mers queries

Observe that knowing the absence of a s-mer allows not only to deduce the absence of the k-mer starting with it: all k-mers containing this s-mer are absents as well. This allows to infer the existence of a stretch of consecutive absent k-mers.

We exploit this simple idea further. If one detects that two absents s-mers are $z+1$ positions away in the query, then any k-mer starting at any position between them is also absent. In the `fimpera` algorithm, if a s-mer is not found during the query, an optimisation consists of searching for the abundance of the s-mer $z+1$ positions further away in the query. If that s-mer is also absent, there is no need to query any s-mer in between.

Thus, `fimpera` only needs to query one s-mer every $z+1$ position as long as the queried s-mers are absent in the `counting AMQ`, effectively saving time. This optimised algorithm is described in supplementary materials, algorithm 3.

2.6 Implementation of `fimpera`

An implementation of `fimpera` is available at `https://github.com/lrobidou/fimpera`. This implementation is specialised for genomic data (i.e. with an alphabet consisting of A, T, C, G) and uses a counting Bloom filter as `cAMQ`. A template mechanism allows the use of any other `cAMQ` provided by the user. In the genomic context, queries consist of fasta or fastq files (gzipped or not), and an option is provided to index and query canonical k-mers only, i.e. the lexicographic minimum between each k-mer and its reverse complements. Options include the k and z values, the size of the filter, and b, the number of bits per abundance count.

As b has a major impact on the final size of the data structure, it is recommended to use low b values (say $b \leq 5$). This limits the maximal stored abundance value to 2^b. In practice `fimpera` includes an option to use any *abundance*

function provided by the user (identity, $\lfloor \log_2 \rfloor$, $\lfloor \log_{10} \rfloor$, range of values, etc...) to compute ranges of abundance per encoded value.

Storing abundances as their $\lfloor \log_2 \rfloor$ values leads to a gain of space (for instance storing up to 64 abundances requires 8 bits, but storing up to $\lfloor \log_2(64) \rfloor$ abundances requires only 3 bits). This is at the cost of a loss of precision for abundances with identical $\lfloor \log_2(64) \rfloor$ values, as this is for instance the case for abundances 5 and 6.

3 Results

3.1 Experimental setup

To the best of our knowledge, no other tool focuses on reducing the false positive rate of existing `cAMQ`, thus we compare `fimpera` results applied on a counting Bloom filter indexing s-mers with the original counting Bloom filter results indexing k-mers. Both methods are using a single hash function. We propose results on biological marine metagenomic data. Parameters used are the default ones: $k = 31$, size of filter of 3.48×10^9 bits, as discussed in section 3.3, using $b = 5$ bits per abundance count, and abundances are stored as their $\lfloor \log_2 \rfloor$ values. We use the default $z = 3$ parameter (unless otherwise stated).

A list of commands for reproducing the results is available here: `https://github.com/lrobidou/fimpera/blob/paper/paper_companion/Readme.md` along with a step-by-step explanation of the output. Executions were performed on the GenOuest platform on a node with 4x8cores Xeon E5-2660 2,20 GHz with 200 Go of memory.

3.2 Metagenomic dataset

We used two fastq files from the TARA ocean metagenomic dataset to show advantages offered by `fimpera` on metagenomic samples. The index was computed from the 2.38×10^8 $31-mers$ present at least twice in an arctic station (accession number ERR1726642) and the query sample was the first 3×10^6 reads from a sample in another arctic station (accession number ERR4691696). Canonical k-mers were considered for this experiment.

3.3 Choice of filters parameters

We propose an experiment in which we apply the `fimpera` approach on top of a `counting Bloom Filter` designed to have 25% of false-positive calls, while using 5 bits per hash value for storing the abundance of indexed k-mers. For indexing 2.38×10^8 $31-mers$ this structure requires 3.48×10^9 bits. Note that when storing the presence/absence of those k-mers in a Bloom filter using the optimal number of hash functions, the expected false positive rate would be 8.7%. We chose to consider a `counting Bloom Filter` that uses a unique hash function. Even if this choice is independent of the `fimpera` approach, it is motivated by the fact

that major tools indexing large sets of k-mers as BIGSI [3], COBS [2], HowDeSBT [5] as well as generic methods for preprocessing k-mers like kmtricks [8] are based on Bloom filters using a unique hash function for performances purposes.

Hence, we propose to index k-mers along with their abundance in a `cBF` of 3.48×10^9 bits using one hash function and storing abundance on 5 bits. We compare the results of queries made against this `counting Bloom Filter` with results of queries made against `fimpera` wrapping that same `counting Bloom Filter`. Unless otherwise specified, all results shown were obtained using $z = 3$. This implies that we compare results of a `cBF` indexing 31-mers, with results of `fimpera` used on a `cBF` with the same sizing, but indexing s-mers of size 28 (31-3).

3.4 Used metrics

To measure the quality of the `fimpera` results and the `cBF` results, we propose three metrics:

- the false positive rate, that provides the probability that the method returns an abundance call > 0 for a k-mer absent from the indexed set.
- the proportion of incorrect abundance, that provides the probability that the method returns the incorrect abundance for a k-mer actually in the indexed set.
- statistics of responses for incorrect abundances calls, that estimate the reported abundance of k-mers whose abundance are incorrectly reported.

3.5 False positive rate analyses

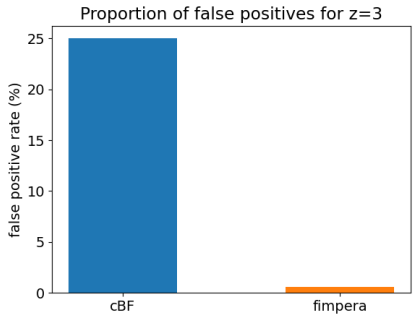

Fig. 1. Proportion of false positive calls without `fimpera` (on a classical `counting Bloom Filter`) and with `fimpera` ($z = 3$).

Results about false-positives obtained with the proposed experiment are shown in Fig. 1. Results about the `cBF` simply confirm the setup, and shows a false

positive rate of 25%. When applying `fimpera`, the false positive rate drops to 0.56%. Among all these `fimpera` false positives, 4.8 % are due to the so-called "construction false positives" (see Section 2.2), thus representing 0.0027% of the total k-mer calls.

It is important to recall that these comparative results were obtained using the exact same amount of space. Hence the `fimpera` approach enabled to yield about 45 times fewer false-positive calls, with no drawback and even saving query time (see Section 3.8).

3.6 Correctness of the reported abundances

In this section, we focus only on true positive calls. Hence, these results do not concern the 25% false-positive calls obtained with the original `cBF`, nor the 0.56 % ones using `fimpera`.

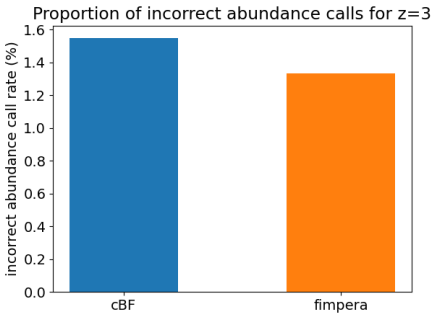

Fig. 2. Proportion of incorrect abundance calls with the original `cBF`, and with `fimpera`

Results comparing the proportion of calls reported with an incorrect abundance among the true positives are shown in Fig. 2. These results show that 1.54 % of true-positive calls are overestimated in the `cBF`, while 1.33 % of true-positive calls are overestimated with `fimpera`. Among the `fimpera` calls estimating an incorrect abundance among the true positives, 83 % are due to the so-called "reconstruction overestimation".

3.7 Distribution of errors in overestimated calls

In this section, we focus only on the wrongly estimated calls among true positives

Results presented Fig. 3 show that, as stated Section 2.2, the erroneous abundance calls are closer to the ground truth with `fimpera` compared to those obtained with the original `cBF`. As seen Fig. 3-right, with `fimpera`, almost all (excepted a few outliers) overestimations are only one value apart from the correct range (the average difference with the correct abundance range is 1.07).

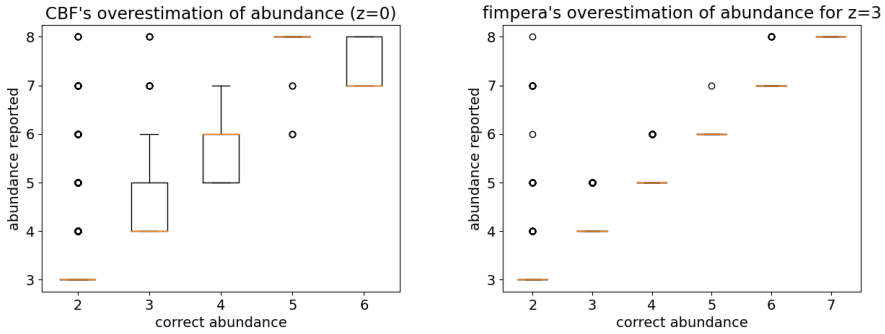

Fig. 3. For true-positive calls with an incorrect abundance estimation: reported abundance with respect to the correct abundance. Left: using the original `cBF`, right: using `fimpera`.

With the original `cBF`, as seen Fig. 3-left, overestimations are more important (1.33 range in average from the ground truth).

3.8 Influence of the z parameter.

The proposed approach requires to set up a unique parameter z that defines the size of the indexed s-mers (recall that $z = k - s$). We propose in this section to assess the impact of this unique additional parameter needed for using `fimpera`.

z	0	3	5	7	9	20
False positive rate (%)	25.00	0.56	0.08	0.02	0.03	99.70
Among which: construction FP (%)	0	0.49	9.64	46.32	67.26	99.99
Incorrect abundance calls (%)	1.55	1.33	1.93	2.99	4.27	42.73
Among which: constr. overestimation (%)	0	83.02	88.65	94.24	94.34	99.85
Query time (s)	808.31	789.56	784.11	785.09	779.42	1596

Table 2. Influence of the z parameter on the quality of the results and on the computation time. "*constr.*" stands for "*construction*". $z = 0$ is equivalent to the original `cBF` results.

Quality of results As shown in Table 3.8, the false positive rate decreases with regard to z and stays low for a wide range of z values (at least from 3 to 9). When using an extreme z value, for instance $s = 20$, the false positive rate is increased up to almost 100%. With $z = 20$, as we use $k = 31$, the size of the s-mers is $s = 11$. When indexing as little as few hundred millions characters, each 11-mer has a great chance to appears by chance in the indexed dataset (it does not occurs with a probability of 10^{-11} when indexing a hundred million characters

on an alphabet of size four). This quasi-random existence of all s-mers generates a huge amount of construction false positives, as seen in the last column. This has also an impact on the running time that nearly doubles, certainly because all queried s-mers are positives, annihilating the s-mer skipping optimisation.

Query time. As mentioned Section 2.5, the `fimpera` approach does not increase the query running. On the contrary, it allows to slightly decrease the running time when z increases as seen Table 3.8.

Default z parameter. Presented results highlight the fact that `fimpera` performances are little impacted by the choice of this parameter. The default z parameter is set to $z = 3$.

4 Conclusion

We presented `fimpera`, a novel computational method to reduce the false positive rate and increase the precision of any counting Approximate Membership Query data structure with no modification of the original data structure. This reduction is obtained without any memory overhead, with no modification of the original data structure, and even with a slight improvement over the query computation time.

Our results showed that when applied on top of a `counting Bloom Filter`, `fimpera` enabled to yields about 45 times fewer false-positive calls than when querying directly a `counting Bloom Filter` of identical size. Moreover, using `fimpera`, abundance errors were slightly less frequent on true positive calls, and finally, those abundance errors were on average 1.07 apart from the ground truth with `fimpera` while they are on average 1.33 apart from the ground truth with the original `cBF`.

Independently from parameters of the used `cAMQ`, `fimpera` requires to set up a unique parameter, z. Fortunately results are highly robust with the choice of z, unless extreme values are chosen. Future work will include a formal analysis of the theoretical limits on the choice of z usage ranges.

We provide a C++ implementation of `fimpera` which enabled us to validate the approach. This implementation can also be used as a stand-alone tool for indexing and querying genomic datasets, and it can be tuned with user-defined parameters and ranges of abundances. The provided github project also proposes all necessary instructions and links to genomic data to reproduce the results.

Acknowledgements. The authors thank Eric Pelletier for his help regarding the usage of the Tara Ocean data sets. This work used HPC resources from the GenOuest bioinformatics core facility (`https://www.genouest.org`). The work was funded by ANR SeqDigger (ANR-19-CE45-0008).

References

1. Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.
2. Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. Cobs: a compact bit-sliced signature index. In *International Symposium on String Processing and Information Retrieval*, pages 285–303. Springer, 2019.
3. Phelim Bradley, Henk C Den Bakker, Eduardo PC Rocha, Gil McVean, and Zamin Iqbal. Ultrafast search of all deposited bacterial and viral genomic data. *Nature biotechnology*, 37(2):152–159, 2019.
4. Carla Cummins, Alisha Ahamed, Raheela Aslam, Josephine Burgin, Rajkumar Devraj, Ossama Edbali, Dipayan Gupta, Peter W Harrison, Muhammad Haseeb, Sam Holt, et al. The european nucleotide archive in 2021. *Nucleic Acids Research*, 50(D1):D106–D110, 2022.
5. Robert S Harris and Paul Medvedev. Improved representation of sequence bloom trees. *Bioinformatics*, 36(3):721–727, 2020.
6. Mikhail Karasikov, Harun Mustafa, Gunnar Rätsch, and André Kahles. Lossless indexing with counting de bruijn graphs. In *International Conference on Research in Computational Molecular Biology*, pages 374–376. Springer, 2022.
7. Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. Kmc 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 2017.
8. Téo Lemane, Paul Medvedev, Rayan Chikhi, and Pierre Peterlongo. kmtricks: Efficient construction of bloom filters for large sequencing data collections. *bioRxiv*, 2021.
9. Camille Marchet, Christina Boucher, Simon J Puglisi, Paul Medvedev, Mikaël Salson, and Rayan Chikhi. Data structures based on k-mers for querying large collections of sequencing data sets. *Genome Research*, 31(1):1–12, 2021.
10. Camille Marchet, Zamin Iqbal, Daniel Gautheret, Mikaël Salson, and Rayan Chikhi. Reindeer: efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*, 36(Supplement_1):i177–i185, 2020.
11. Lucas Robidou and Pierre Peterlongo. findere: fast and precise approximate membership query. In *International Symposium on String Processing and Information Retrieval*, pages 151–163. Springer, 2021.
12. Brad Solomon and Carl Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology*, 34(3):300–302, 2016.
13. Shinichi Sunagawa, Silvia G Acinas, Peer Bork, Chris Bowler, Damien Eveillard, Gabriel Gorsky, Lionel Guidi, Daniele Iudicone, Eric Karsenti, Fabien Lombard, et al. Tara oceans: towards global ocean ecosystems biology. *Nature Reviews Microbiology*, 18(8):428–445, 2020.
14. Katarzyna Tomczak, Patrycja Czerwińska, and Maciej Wiznerowicz. Review the cancer genome atlas (tcga): an immeasurable source of knowledge. *Contemporary Oncology/Współczesna Onkologia*, 2015(1):68–77, 2015.

S1 Supplementary Materials

These supplementary materials propose a detailed description of the proposed algorithms.

S1.1 query algorithm

A non-optimised algorithm of `fimpera`'s query is shown in Algorithm 1. This algorithm takes a queried sequence q, a `counting AMQ` indexing s-mers, and parameters k and z. It returns a vector of integers such that: $\forall i \in [0, |response|]$, $response[i]$ is the abundance of the k-mer starting at position i in the query.

Algorithm 1 `fimpera`'s query

1: **procedure** QUERY($q \in \Sigma^*$; cAMQ indexing s-mers; k and z in $\mathbb{N}^+$ ($|q| \geq k, z \leq k$))
2: $s \leftarrow k - z$ ▷ may be zero
3: $smer_s_{abundances} \leftarrow emptyVector(0)$
4: # Store $s_{abundance}$ of all s-mers:
5: **for** i in $[0; |q| - s]$ **do**
6: $ab \leftarrow s_{abundance}$ of the s-mer starting pos i in q (using the `counting AMQ`).
7: add ab in the $smer_s_{abundances}$ vector
8: **end for**
9: # Compute all k-mers abundances from s-mer $s_{abundances}$:
10: $response \leftarrow emptyVector(|q| - k + 1)$
11: **for** i in $[0; |q| - k]$ **do**
12: add $\min_{j \in [i, i+z]}(smer_s_{abundances}[j])$ in the $response$ vector
13: **end for**
14: **return** $response$
15: **end procedure**

Algorithm 1 is not optimal. Line 12 it computes the minimal value of a range of z consecutive integers taken from a vector of integers. At each iteration of a *for loop* this range is shifted by one. An optimization, presented in the next Section, enables to compute all these minimal values in linear time and with zero memory allocation.

S1.2 Sliding window minimums

Algorithm 2 sliding_minimum_window

procedure SLIDING_MINIMUM_WINDOW(vector v of positive integers; length of window w ($|v| \geq w, w > 1$))
$nbWin \leftarrow \lfloor size(v)/w \rfloor$
$nb_elem_last_window \leftarrow |v| \mod w$
$min_left \leftarrow v[0]$ ▷ start computation of min_left (See Section 2.5)
for i in $[0; w - 1]$ **do**
$min_left \leftarrow min(min_left, v[i])$
end for
for i in $[0; nbWin - 2]$ **do** ▷ for every window excluding the last one

```
        start_window ← i ∗ w
        for index in [start_window + w − 2; start_window] (decreasing order) do
            v[index] ← min(v[index + 1], v[index])    ▷ compute min_right, directly in v
        end for
        for j in [0; w-1] do
            #sliding_minimum is min(min_left, min_right(=v)):
            v[start_window + j] ← min(v[start_window + j], min_left)
            #update min_left (if a new fixed window starts, reset min_left):
            if j == 0 then
                min_left ← v[start_window + w]
            else
                min_left ← min(min_left, v[start_window + w + j])
            end if
        end for
    end for
    # Computation for the last window is not described here for the sake of simplicity
    # remove last w − 1 elements from v:
    for i in [0; w − 2] do
        v.pop_back()
    end for
    return v
end procedure
```

S1.3 Skip streches

In this section we show the entire algorithm of `fimpera` (algorithm 3), including the optimisation consisting in skipping stretches of absent k-mers. The skip optimisation occurs line 32: if a negative s-mer is called, the algorithm jumps $z+1$ position away in the sequence, probing for another absent s-mer. A positive answer will trigger line 18, backtracking z positions backward. We keep track of the fact that we are currently skipping s-mers via the *extending_stretch* flag.

Algorithm 3 `fimpera`'s query

```
1: procedure QUERY(q ∈ Σ*; cAMQ indexing s-mers; k and z in ℕ+ (|q| ≥ k, z ≤ k))
2:     s ← k − z
3:     response ← emptyVector(size − K + 1)
4:     stretchLength ← 0
5:     j ← 0                                    ▷ Current position in the query
6:     extending_stretch ← true
7:     previous_answers ← emptyVector(0)
8:     while j < |q| − k + 1 do
9:         smer ← smer_starting_at_position_j_in_q
10:        amq_answer ← s_abundance_of_smer_in_cAMQ
11:        if amq_answer > 0 then
12:            if extending_stretch then
13:                previous_answers.push_back(amq_answer)
14:                stretchLength ← stretchLength + 1
```

```
            j ← j + 1
        else
            extending_stretch ← True
            j ← j − z
        end if
    else
        if stretchLength > z then
            start_of_stretch ← j − stretchLength
            offset ← 0
            for minimum : sliding_window_minimum(previous_answers, z+1) do
                response[start_of_stretch + offset] ← minimum
                offset ← offset + 1
            end for
        end if
        previous_answers ← emptyVector(0)
        stretchLength ← 0
        extending_stretch ← false
        j ← j + z + 1
    end if
end while
if stretchLength > z then
    start_of_stretch ← |q| − k + 1 − stretchLength;
    offset ← 0
    for minimum : sliding_window_minimum(previous_answers, z + 1) do
        response[start_of_stretch + offset] ← minimum
        offset ← offset + 1
    end for
end if
return response
end procedure
```