Search engine for genomic sequencing data

Lucas Robidou

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par
Lucas ROBIDOU

Moteur de recherche pour données de séquençage génomique

Thèse présentée et soutenue à Rennes, le 21 septembre 2023
Unité de recherche : Équipe GenScale, Univ Rennes, Inria, CNRS, IRISA

Rapporteurs avant soutenance :

Paola BONIZZONI Full Professor, Università di Milano-Bicocca
Nadia PISANTI Associate Professor, University of Pisa

Composition du Jury :

Président :	Jacques NICOLAS	Directeur de recherche, Inria Rennes
Examinateurs :	Paola BONIZZONI	Full Professor, Università di Milano-Bicocca
	Nadia PISANTI	Associate Professor, University of Pisa
	Lois MAIGNIEN	Assitant Professor, UBO Brest
	Giulio ERMANNO PIBIRI	Assistant Professor, Ca' Foscari University of Venice
	Jacques NICOLAS	Directeur de recherche, Inria Rennes
Dir. de thèse :	Pierre PETERLONGO	Directeur de recherche, Inria Rennes

Acknowledgement

First and foremost, I would like to thank Paola Bonizzoni and Nadia Pisanti for accepting to be reviewers of my thesis. I want to thank Lois Maignien, Giulio Ermanno Pibiri and Jacques Nicolas for accepting to be part of my jury. Thank you all for the interest you show in this work.

Je remercie également Pierre Peterlongo, mon encadrant, pour m'avoir.. Eh bien, encadré. Tes conseils aident à prendre de la hauteur[1] et sont toujours bienvenus. En particulier, j'ai beaucoup apprécié ta façon de donner des conseils ("*Je* pense que..."), ça aide à s'approprier la recherche effectuée.

Merci beaucoup à tout Symbiose[2] : l'environnement de travail est exceptionnel. Je pense notamment à Marie, sans qui nous serions bien incapables de faire quoi que ce soit !

Merci aux collègues pour les discussions diverses et variés[3]. Que ce soit sur le plan scientifique (sur le choix du jury de thèse par exemple) ou plus technique (LaTeX, rust, LaTeX et LaTeX), les discussions n'ont pas manquées d'intérêt.

J'en profite aussi pour remercier mes professeurs, sans qui je n'aurais pas été aussi loin :

Merci à M. Chaussard, qui m'a fait me plonger dans le C, les erreurs de segmentation et valgrind (dans cet ordre).

Merci à M. Létocart, pour m'avoir enseigné l'architecture des ordinateurs, le cours le plus important pour des ingénieurs en informatique, selon moi. Merci aussi pour l'optimisation linéaire et combinatoire. Avant que j'explique aux gens comment bien faire leur valise, ça leur prenait 20 minutes, et après ça m'en prends 15.

Merci à Mme Sirieix pour m'avoir montré tout ce qu'on peut faire avec un peu de Python et d'imagination. J'aurai aimé être aussi bon en mathématiques qu'en informatique... On va dire que ça ne s'est pas (trop) vu.

Merci à tous les autres professeurs que j'ai eu durant ces deux ans de prépa pour m'avoir tant appris.

1. Il fallait bien une blague sur l'escalade !
2. Mais quand même un peu plus à Genscale. Juste un peu.
3. Et les jeux de mots avariés.

Merci à M. Trellu pour m'avoir rappelé continuement, à chaque cours, pourquoi j'aimais la science.

Merci à mon professeur de latin de lycée, sans qui le nom des contributions présentées dans ce documents seraient certainement beaucoup plus simple.

J'en profite également pour remercier mes amis et mes proches :

Merci à Brieuc, Nicolas et Guillaume pour m'avoir supporté aussi longtemps. Surtout Nicolas, le pauvre, ça doit bien faire 25 ans non ?

Merci à Thibaud et Sébastien pour ces moments en école d'ingénieur. En presque 9 mois, on aura fait un beau bébé[4] !

Merci à Olivier pour ces heures de jeu, ça aide beaucoup à décompresser.

Merci à Victor pour le petit (hum) grain de folie dans le labo.

Merci à Kerian pour ces bons moments ciné et jeux ! Et les chats ! (Et les chats, et les chats, et aussi les chats...)

Merci à Meven pour les nouilles, ce plat du nouvel an, les ramens, ce bibimbap...

En particulier, merci à Garance, pour... tout. Voilà, ça fais 10 minutes que je sèche sur comment te remercier, alors je vais faire court : merci du fond du cœur.

Merci surtout à Lo pour tous ces moments partagés ensemble. Je crois que tu l'auras deviné, j'ai un peu de mal avec cet exercice de remerciement !

Merci à Youen et Béatrice pour me supporter encore depuis toutes ces années !

Merci à Arthur pour avoir été là, particulièrement durant ces deux ans de prépa.

Merci beaucoup (beaucoup beaucoup) à ma famille pour leur soutien. J'ai un peu peur que vous vous ennuyiez si vous insistez pour venir à la soutenance... Courage, il y aura du gâteau à la fin !

4. OK, en PHP :)

Pour finir, une sélection de phrases rencontrées ici et là :

«Je pense être plutôt neurotypique...»
S'éloigne en sautillant.
«Coin Coin Coin Coin Coin...»

Suite à une discussion sur la synesthésie[5] entre collègues :
«Les mois ont une couleurs mais c'est tout, arrêtez vous
[...]
jvais changer de sujet de thèse
"Mes collègues et leurs trucs chelou, dans leurs têtes, liés à la nourriture, aux chiffres et aux lettres" le titre peut s'allonger au besoin»

Ma première note en prépa :
«1».
Paniqué, je me tourne vers mon voisin pour savoir si c'est sur 10 ou sur 20. Je regarde sa copie :
«-2».

5. phénomène neurologique touchant environ 4% de la population, par lequel plusieurs sens sont associés durablement

RÉSUMÉ EN FRANÇAIS

Introduction

L'ADN est une molécule retrouvée chez toutes les espèces vivantes connues. Cette molécule contient les informations nécessaires aux être vivants pour se nourrir, croître et se reproduire. Sa structure, mise en évidence en 1953, se compose d'un enchaînement de quatre blocs élémentaires, appelées "bases" (cytosine [C], guanine [G], adenine [A], et thymine [T]). Chaque individu possède, dans chacune de ses cellules, une ou plusieurs molécules d'ADN, dont l'enchaînement est propre à chaque individu.

Chaque molécule d'ADN peut être représentée sous forme de long texte[6] en utilisant un alphabet de quatre lettres. Ce texte est un élément clef pour comprendre le fonctionnement de l'être vivant dont il provient ; il est donc d'un intérêt crucial pour les biologistes. Le domaine de la science qui étudie le traitement de ces séquences et son automatisation est appelé la *bioinformatique des séquences*.

L'acquisition de courts fragments (environ une centaine de lettres) de ce texte à partir d'une molécule d'ADN est permise dans les années 1970, notamment par la méthode développée par Sanger [1] en 1977. Chaque fragment est appelé "lecture".

L'acquisition des séquences d'ADN s'est ensuite améliorée avec le développement des séquenceurs de deuxième génération autour des années 2000, qui augmentent la taille des lectures à quelques centaines de bases. Ils permettent aussi la parallélisation de la lecture d'ADN, abaissant drastiquement le coût de ces lectures et permettant de séquencer des millions de bases d'un coup [2], diminuant le temps de séquençage.

La troisième génération de séquenceurs permet de lire de plus longues séquences, au prix d'une augmentation de la fréquence d'erreurs dans les lectures (insertion ou délétion de base par rapport à la molécule lue, remplacement d'une base par une autre...).

Ces lectures générées par les séquenceurs sont ensuite stockées sur un support informatique. Chaque expérience de séquençage (contenant plusieurs milliers, voir millions, de lectures) est représentée dans un jeu de données. Le nombre de ces jeux augmente rapidement, si bien que l'espace requis pour tous les stocker augmente exponentiellement

6. Il faut plus de 10^9 lettres pour représenter le "texte" dans chaque cellule d'un être humain.

avec les années. Il y a désormais tellement de jeux de données qu'à partir d'une séquence, il est impossible de retrouver dans quel(s) jeu(x) elle pourrait apparaître.

Il est possible de comparer la situation actuelle des bioinformaticiens avec celle des internautes avant l'apparition des moteurs de recherche : une manne de contenu est publiquement disponible, mais il n'existe aucun moyen de retrouver où une information d'intérêt pourrait apparaître.

Construire un tel moteur de recherche pour les lectures générées par des séquenceurs est le but de la recherche présentée dans ce manuscrit.

À partir d'une séquence ADN, appelée requête, un moteur de recherche permet de retrouver dans quel(s) jeu(x) cette requête peut apparaître.

Intuitivement, pour chaque requête, il suffirait de lire l'entièreté des données disponibles publiquement et de les comparer avec la requête. Or, comparer une requête avec chaque lecture de chaque jeu de données prendrait des années, voir des siècles. Ce processus serait si long que le résultat de la requête finirait par devenir obsolète avant la fin du processus ! Un moteur de recherche doit donc trouver les jeux de données similaires à une requête, mais sans les lire.

Les moteurs de recherches travaillant sur des langages naturels utilisent la notion de "mot" pour représenter leurs jeux de données. Dans notre contexte, un mot est une suite de caractères de taille arbitraire, mais finie.

Dans le cas des langues indo-européennes, il est usuel de délimiter les mots à l'aide d'un caractère spécifique (une espace). Or, les lectures d'ADN se représentent sur un alphabet de seulement quatre lettres. Il n'y a donc pas de lettre spécifique pour découper une lecture en mots. La découpe en mot s'effectue donc de la façon suivante : **1/** La taille des mots, notée k, est fixée en amont du découpage. **2/** Chaque lecture est découpée en tout les mots de taille k qu'elle contient. En numérotant les lettres d'une lecture à partir de 0, le premier mot contient les lettres de la position 0 à la position $k-1$, le second de la position 1 à k, etc.

Ces mots de taille fixes sont appelés k-mers. Ils sont une composante essentielle dans les outils travaillant sur l'analyse des séquences d'ADN.

Dans notre cas d'utilisation, ces mots sont ajoutés dans une structure de données, appelée "filtres `AMQ`" (de l'anglais "Approximate Membership Query", littéralement "Requêtes d'appartenance approchée"). Ces filtres `AMQs` permettent d'indexer des éléments

(dans notre cas : des k-mers) en utilisant peu d'espace mémoire. En contrepartie du gain de place proposé, seules deux opérations sont possibles sur les filtres `AMQs` :

- ajouter un élément dans le filtre
- demander si un mot est présent dans le filtre. La réponse à cette demande est sujette à des faux positifs : certains éléments absent sont répondus présents. La proportion d'éléments absents trouvés dans un filtre `AMQ` est appelé son taux de faux positif, noté FPR_{AMQ} ($FPR_{AMQ} \in [0; 1]$) dans ce manuscrit.

Le taux de faux positif non nul est la principale limitation des filtres `AMQs`. Il existe plusieurs implémentations de filtre `AMQ`, chacune proposant un compromis entre espace mémoire utilisé et taux de faux positif.

Certaines structure similaires, appelées "filtres `cAMQ`", permettent de stocker une approximation du nombre de fois où chaque mot apparaît dans un jeu de données (son comptage). Avec un filtre `cAMQ`, les requêtes sur les comptages peuvent renvoyer des surestimations en plus des faux positifs.

En pratique, un moteur de recherche fonctionne en deux temps :
1/ indexation : chaque jeux de données à indexer est parcouru, découpé en mots et ces mots sont ajoutés[7] dans un filtre `AMQ` (un par jeu)
2/ requête : la requête est découpée en mots, qui sont recherchés dans le filtre `AMQ` de chaque jeu de données. La proportion de mots trouvés dans chaque filtre `AMQ` renseigne sur la probabilité de la présence de la requête dans le jeu de données associé.

Les faux positifs biaisent la proportion des mots trouvés dans un filtre. La surestimation de cette proportion impacte directement le résultat des requêtes.

> La réduction de ces faux positif et des surestimations induites par les filtres (c)-`AMQ` est l'axe majeure de cette thèse.

Contribution 1

Dans notre première contribution, nous nous proposons de réduire le taux de faux positifs des filtres `AMQs` indexant des k-mers extraits de séquences. Ces séquences peuvent être

7. Potentiellement après une étape de filtration (par exemple, pour ne pas tenir compte des mots présent une seule fois : ces mots contiennent en effet probablement des erreurs de séquençage).

d'origine biologique ou non : notre contribution s'applique également à des phrases en langage naturel.

Notre stratégie de réduction des faux positifs, appelée `findere`, repose sur l'observation suivante :

- Si un k-mer est présent dans une séquence, alors toutes les sous-séquences du k-mer sont aussi présentes dans cette séquence.
- Par contraposée, si une des sous-séquence du k-mer n'est pas présente dans une séquence, alors le k-mer est aussi absent de cette séquence.

Par exemple, si une chaîne de caractère contient le 9-mer "recherche", alors les 7-mers "recherc", "echerch", "cherche" sont aussi présente dans cette chaîne. À contrario, si le 7-mer "recherc" est absent d'une chaîne de caractère, alors le 9-mer "recherche" est absent également.

`findere` repose sur ce principe. Au lieu d'indexer des mots de taille k, `findere` repose sur l'indexation de mots de taille s (s comme "short", avec $s \leq k$). Lors de la requête, un k-mer est considéré comme présent si et seulement si tout ses s-mers sont présents.

Afin de faciliter la lecture, on notera $z = k - s$. Un k-mer est donc constitué de $z + 1$ s-mers.

Il se peut qu'un k-mer absent soit constitué de s-mers présents, induisant un faux positif dit *de construction*. Si, dans une requête, un k-mer est absent et est constitué de $z+1$ s-mers absents, alors il faut $z+1$ faux positifs pour considérer, à tort, ce k-mer comme étant présent. La probabilité pour ce k-mer d'être faux positif est donc ${FPR_{AMQ}}^{z+1}$, ce qui diminue de manière exponentielle avec z. Dans le cas général, des k-mers absents peuvent contenir des s-mers présents, ce qui diminue le nombre de tests disponibles pour rejeter la présence de ces k-mers.

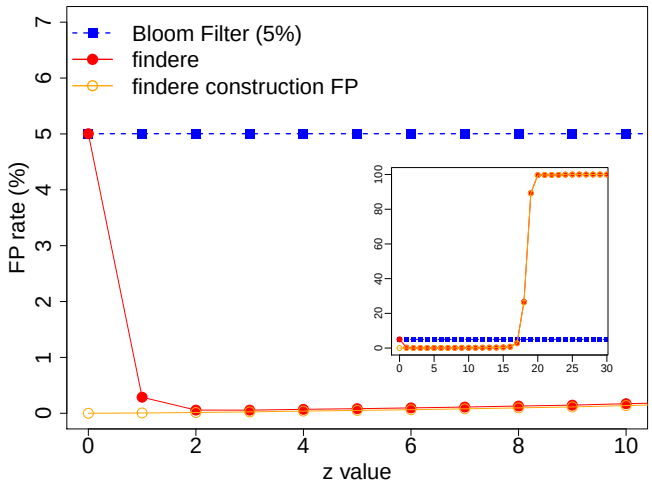
Nous avons testé notre approche sur deux types de données :

- des données génomiques (indexant et requêtant deux jeux distincts extrait de HMP [3], "Human Microbiome Project"), contenant respectivement 4.2 et 3.2 millions de lectures, d'une taille moyenne de respectivement 92 et 96 lettres
- deux corpus de textes en langage naturel (tirés de Wikipédia), chacun constitués de

10^8 31-mers

Données génomiques

Dans chacune des expériences, nous avons indexé des 31-mers dans un filtre de Bloom (l'un des filtres `AMQ` les plus utilisées en bioinformatique des séquences). Les résultats de l'expérience utilisant des données génomiques sont donnés en Figure 1.



Taux de faux positif en fonction de z, pour $k = 31$

FIGURE 1 : Comparaison des taux de faux positifs obtenus lors de l'indexation puis de la requête des jeux de données HMP. Un filtre de Bloom indexe des 31-mers, avec un taux de faux positifs $FPR_{\texttt{AMQ}} = 5\%$ (carré pleins bleus, indépendant de la valeur de z). En utilisant les mêmes paramètres pour la construction du filtre de Bloom, `findere` a été lancé pour différentes valeurs de z (et donc de s), donnant un taux de faux positifs représenté par des cerces rouges pleins. Les cercles orange représentent le taux de faux positifs de construction. La figure représentée est un zoom sur les valeurs des z que nous recommandons (en particulier $z \in [2, 5]$). La figure dans l'encadré montre également les valeurs obtenus pour des valeurs de z plus importantes. Ici, avec $k = 31$ nous déconseillons l'utilisation de valeurs de s plus petite que 15, donc de z plus grande que 16.

Dans le contexte génomique, avec $z = 3$, `findere` réduit le taux de faux positif de deux ordres de grandeur. Ces résultats se retrouvent dans le cas du langage naturel.

Une des limites de notre méthode est que lorsque le taux de faux positif du filtre de Bloom est choisi sous le taux de faux positif de construction, `findere` augmente le

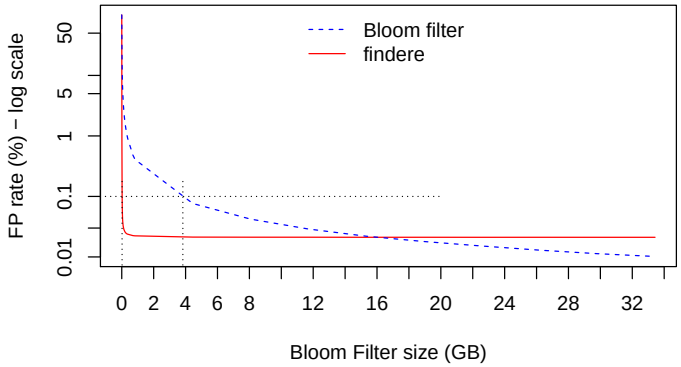
taux de faux positif. En pratique, pour des données génomique, le taux de faux positif de construction est de 0.025% avec les paramètres recommandés, alors que le taux de faux positifs par défaut des méthodes d'indexation varie généralement entre 5% et 25%. Ainsi, ce désavantage de notre approche n'a, en pratique, pas d'impact.

Application au langage naturel

Nous avons appliqué notre méthode sur deux corpus de texte en langage naturel, tous deux tirés de Wikipédia et de taille similaires.

Nous avons extraits les 31-mers des deux jeux (environ 10^8 31-mers dans chacun des jeux) et avons indexé ceux tirés du premier jeu dans deux filtres de Bloom de même taille. Nous avons appliqué `findere` sur un des deux filtres, en choisissant $z = 3$. Nous avons ensuite requêté ces deux filtres avec les 31-mers du second jeu afin de déterminer leur taux de faux positifs. Le procédé a été répété en faisant varier la taille des filtres.

Les résultats de cette expérience sont donnés en Figure 2.



Taux de faux positif en fonction de la taille du filtre de Bloom, corpus en langage naturel, $k = 31$ et $z = 3$

FIGURE 2 : Comparaison des taux de faux positifs obtenus lors de l'indexation puis de la requête des jeux de données en langage naturels tirés de Wikipédia. La ligne en pointillée correspond à un taux de faux positifs de 0.1%.

Dans le contexte du langage naturel, pour un taux de faux positif de 0.1%, `findere` réduit l'espace nécessaire pour représenter un filtre de Bloom. En effet, l'espace nécessaire pour représenter un filtre de Bloom est de 3.38 Go sans `findere`, tandis qu'avec `findere` cet espace nécessaire est de 0.023 Go pour $z = 3$.

Cette expérience montre également une limite de `findere` : lorsque la mémoire allouée au filtre dépasse un certain seuil, appliquer `findere` sur ce filtre en augmente le taux de faux positifs.

Un texte en langage naturel est constitué de mots de taille généralement inférieure à 31 lettres. Il n'est pas rare qu'ils se répétent plusieurs fois au sein d'un même texte, dans des phrases différentes.

Cela influence la probabilité d'occurrence des faux positifs de construction, puisque qu'un 31-mer donné est plus susceptible d'avoir des portions communes avec d'autres 31-mers distincts dans un jeux de données en language naturel que dans un jeux de données métagénomique. Ceci pourrait expliquer pourquoi la limite de `findere` est atteinte pour un filtre de 15 Go dans ce contexte, mais pas dans le contexte de données génomiques.

Conclusion

`findere` a permis la diminution de taux de faux positifs des filtres de Bloom construits par kmtricks [4], ce qui a permis de mettre à disposition du grand public une interface web permettant de chercher une séquence d'intérêt parmi les données de *Tara* oceans [5]. Cette interface est disponible à l'adresse suivante : `https://ocean-read-atlas.mio.osupytheas.fr/`.

Contribution 2

Notre seconde contribution, `fimpera`, a consisté à réduire la surestimation induite par les filtres `cAMQs` en généralisant le principe exposé plus haut :

- Si un k-mer est présent x fois dans une séquence, alors toutes les sous-séquences du k-mer sont aussi présentes au moins x dans cette séquence.
- Par contraposée, si une des sous-séquence du k-mer est présente au plus y fois dans une séquence, alors le k-mer est aussi présent au plus y fois dans cette séquence.

Pour prendre en compte le comptage de chaque k-mer, nous proposons d'indexer des s-mers avec leurs compte, puis de considérer que le compte d'un k-mer est le minimum du compte de ses s-mers.

Nous proposons deux façons de procéder :

- Soit associer chaque s-mers à son propre comptage dans le filtre `cAMQ`.
- Soit associer dans le filtre `cAMQ` chaque s-mer avec le maximum des comptes des k-mer dans lequel il apparaît.

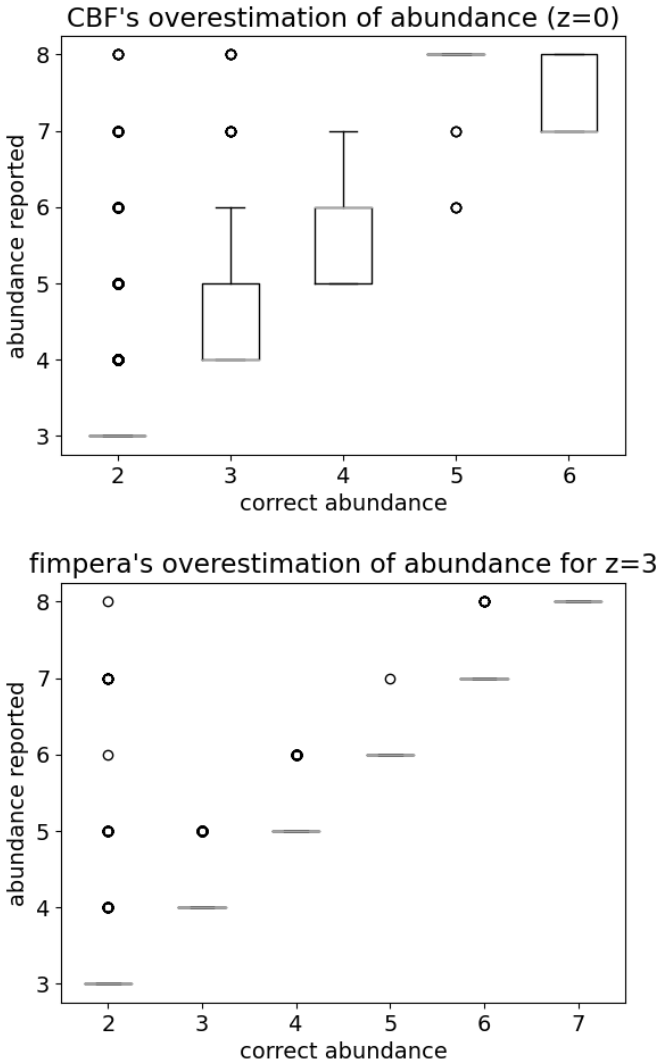
La deuxième manière de procéder permet de ne pas surestimer un k-mer lorsqu'il partage ses s-mers avec d'autres k-mers distincts, mais demande de modifier l'étape d'indexation dans le filtre `cAMQ`, cassant la compatibilité avec les systèmes existants.

Appliquée à des jeux génomiques venant de *Tara* océans [5], cette méthode a permis de réduire les surestimations induites par l'utilisation d'un filtre de Bloom avec comptage, un des filtres `cAMQ` le plus utilisé, comme montré Figure 3.

Une contribution indépendante (développée dans le cadre de `fimpera`), nous a amené proposer un algorithme pour calculer le minimum de chacune des fenêtres glissantes de taille fixées d'un vecteur. L'atout majeur de cette méthode est d'éliminer le besoin de recourir à des allocations dynamiques de mémoire, ce qui en fait la méthode la plus rapide connue pour réaliser cette tâche.

Conclusions

Ces deux contributions améliorent l'utilisation de structures de données très utilisées en bioinformatique des séquences. Nos pensons qu'elles seront utiles pour réduire le taux de faux positifs et / ou de surestimations dans les méthodes reposant sur des filtres (c)-`AMQs`. Ces contributions ont d'ailleurs été mises à profits dans kmtricks [4], puis plus tard dans kmindex [6], des outils de construction et de requête de filtres de Bloom indexant des k-mers avec et sans comptages associés.



Logarithme de l'abondance de 31-mers trouvés dans un filtre en fonction du logarithme de leur abondance réelle.

FIGURE 3 : Comparaison de la surestimation induite lors de l'indexation puis la requête de jeux génomiques venant de *Tara* océans [5]. Un filtre de Bloom avec comptage indexe des 31-mers ; `fimpera` est appliqué sur un filtre de même taille ($z = 3$). Chacune des deux méthodes indexe et requête le logarithme de l'abondance des 31-mers. Haut : filtre de Bloom avec comptage, bas : `fimpera`.

Table of Contents

Chapter 1

Introduction

1.1 Biological context

Biology is a natural science whose subject is life. As such, a motive of biologists is to understand beings of many kinds: viruses, bacteria, plants, yeast, animals... Being such an extended field, biology helps to understand a large scope of events, such as how cells develop, how climate change impacts organisms [7], how diseases develop, manifest and spread, the inner process of viruses [8], [9], etc. To properly grow, move and reproduce, each organism holds a detailed description of its structure and processes. That description is called "genetic information" and is stored in their DNA. DNA is a macromolecule that holds genetic information through a sequence of four molecules, called "bases", each represented by a letter: cytosine [C], guanine [G], adenine [A], and thymine [T]. Each DNA molecule can thus be considered as a sequence of letters, called a **DNA sequence**.

As DNA sequences contain every genetic information of any given organism, they provide tremendous information on the organism they come from. Therefore, DNA sequence analyses are of high interest to biologists. As such, this work focuses on these sequences of letters.

For biological reasons, each DNA sequence is duplicated on two strands that retain the same genetic information. The two strands are chemically linked to one another: each base on one strand is matched with another base on the other strand. Each base can be linked to another specific base, called its complement. An [A] (resp. [C], [T], [G]) on a strand is always matched by its complement, [T] (resp. [G], [A], [C]). See Fig. 1.1 for an illustration. The reading direction of two strands bound together is reversed. For instance, if a strand contains the sequence "ATCG", the other strand contains the sequence "CGAT" (on Fig. 1.1, "ATCG" is on the left strand if you read top-down, while "CGAT" is on the right strand if you read bottom-up). Given a DNA sequence, we refer to the sequence on the other strand as its "reverse complement". As a sequence and its reverse complement hold the same genetic information, tools that manipulate DNA sequences must consider

those sequences as equivalent. In the context of this work, this is achieved through the concept of canonicity, explained in Section 1.5.

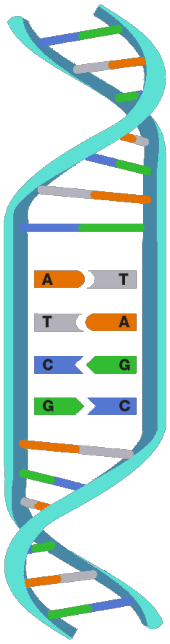


Figure 1.1: A representation of a portion of a DNA molecule. Each base on a strand is matched with its corresponding complement on the other strand. Source: NIH [10]

Note:
From now on, I may use examples using English sentences instead of DNA sequences, as it is easier to spot repetitions and differences among two English sentences rather than in two DNA sequences.
For instance, try spotting the difference between:

- "Jokes about German sausage are the wurst."
- "Jokes about German sausage are the worst."

Now try with:

- "ACATACTCGACTGAATATATTAGCTGAGATAAGTGGTAGAT"
- "ACATACTCGACTCAATATATTAGCTGAGATAAGTGGTAGAT"[1].

1. the thirteenth letter was changed from G to C

Even if both those English sentences and those DNA sequences have the same length, it should have been easier to spot the difference in English. Some concepts can not be represented using English sentences (e.g. the concept of "complementing a letter" does not exist in English). In such a case, I will provide examples using DNA sequences.

Of course, before analyzing sequences, biologists must first acquire them. Reading a DNA sequence requires specific tools, called sequencers, that are described in the following section.

1.2 Sequencers

Sequencers are machines that take DNA molecules as input and produce the corresponding sequences in a text file as output.

The ability to sequence DNA is currently having a transformative impact on many scientific fields, with DNA sequencing providing means to detect a wide range of diseases. By having access to the DNA sequence of a patient, scientists can detect genetic disorders using blood samples and may even explore potential cures. These analyses can be performed even before birth, such as in the case of Down syndrome which can be detected from the expectant mother's blood [11]. In France, sequencer-based tests are allowed to complement traditional tests for this purpose [12]. The range of applications of sequencers is amazingly wide. For instance, they can even detect skin diseases many years after the patient's death; the blood of Jean-Paul Marat, a French revolutionary, was analyzed two centuries after his death, revealing that he had seborrheic dermatitis [13]. Sequencers can also help to detect antibiotic resistance [14] and prevent giving patients unsuitable drugs. The applications of sequencers extend beyond medicine to fields such as forensics [15], biotechnology, agronomy [16], ecology, and phytoplankton distributions in the oceans [17], among other examples.

Sequencers have become increasingly portable, allowing them to be utilized in hospitals and farms alike. However, it should be noted that sequencers do not provide a completely accurate base sequence from DNA molecules, as errors and biases can be introduced during the sequencing process.

Sequencers only provide small subsequences of DNA, called "reads", composed of typically hundreds or thousands of letters each (see a representation in Figure 1.2). This

is only a very small fraction of a molecule: a human DNA molecule contains up to hundreds of millions of letters. Moreover, the original location of reads on DNA is unknown (a read can be thought of as a random sample of the sequenced DNA). Fortunately, reads may overlap, allowing to partially reconstruct the original DNA sequence.

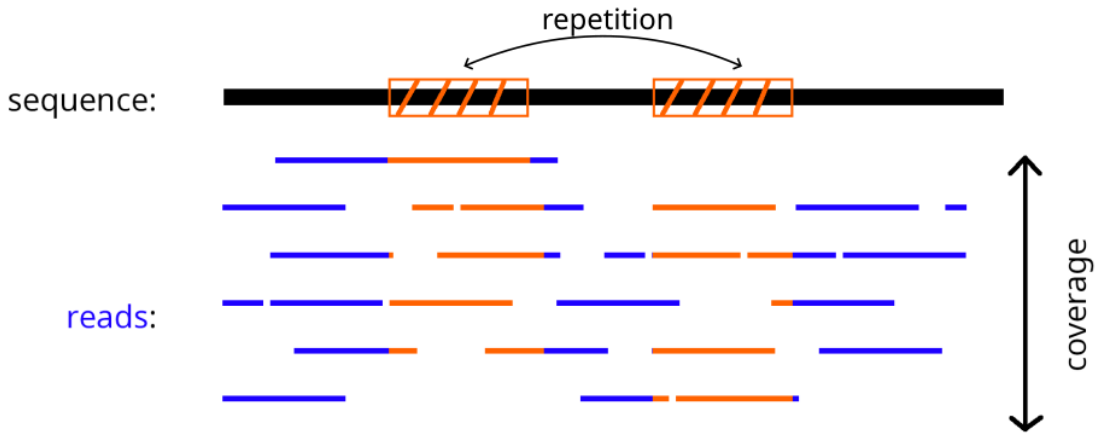


Figure 1.2: Representation of reads given by a sequencer over an input sequence. Orange regions represent a repetition, which would be found in the reads over these regions as well. The average number of reads covering each base is called the "average coverage". The topmost read covers both the repetition and its left and right context. Note that over a human genome, depending on the sequencing method, reads would appear much smaller and there would be millions of them.

For instance, consider two English sentences:

- "*in the south, the weather is always sunny*"
- "*in the north, the weather is always awful*"

If you extract only short fragments from these sentences, you may end up with the following reads:

- "*in the south, the weather is*"
- "*the weather is always awful*"
- "*in the north, the*"
- "*is always nice*"

Given those reads, you may reconstruct the sentence "*in the north, the weather is always awful*".

However, you may also incorrectly reconstruct the sentence "*in the south, the weather is always awful*". That is because the two original sentences share a repetition ("*th, the weather is always* ") and no read covers the entire repetition and both its left and right context. The difficulty to obtain a complete sequence from long or highly repeated genomes leads to tremendous implications for tools dealing with reads from these genomes. A basic issue is the fact that repetitions introduce ambiguity over the original sequence. Given a read contained in a repeated region in the DNA, it is impossible to guess which region it overlaps.

In practice, among the huge number of reads produced by the sequencer, one read may cover both the words "*south*" and "*sunny*", allowing you to remove the ambiguity. Hence, the longer the repetition, the longer the read needs to be to remove the ambiguity.

Given the huge size of a DNA molecule and the relatively small length of reads, there is a probability that a position in the DNA sequence will not be covered by any read. Fortunately, most recent sequencers output up to hundreds of millions of reads per run, mitigating this risk. In practice, a base is covered by multiple reads. The number of reads a base is covered by is called its "coverage". The more reads a sequencer outputs, the higher the average coverage of bases. However, doing so increases the cost of sequencing, the time required to perform both the sequencing and the post-processing, and the disk usage required for storing reads. The amount of space required to store all the publicly available reads is becoming more and more important, as stated in Section 1.4.

Sequencers can not tell from which DNA strand a read comes from and can introduce mistakes on reads (erroneously introducing or removing bases, or mistakenly replacing a base with another one). A high coverage ensures that each base is correctly sequenced on some reads, allowing to post-correct erroneous bases.

There are three generations of sequencers: each generation corresponds to specific sequencing technologies. Each technology yields different throughput and precision. Those generations are described in the following section.

First generation of sequencers

The first generation of DNA sequencers was developed around 1970 [18] and later by Sanger [1]. Another method was developed by Maxam and Gilbert [19]. These methods rely on chemical reactions to extract the DNA sequences, e.g. by cutting the DNA into

fragments at each repetition of a base in Maxam's process (the length of fragments thus indicates the position of the aforementioned base). Those methods are difficult to automatize. Furthermore, Maxam's one requires hazardous chemicals and is now almost unused. Sanger's method allows sequencing up to a hundred bases per hour[2]. The size of the reads is about a few hundred bases long but virtually error-free (about 99.99% of correct bases [20]). As such, the current usage of Sanger's method is to read DNA sequences with a low error rate when the sequencing speed (or read size) is not an important criterion, since this method is outperformed by the following generations.

Next generation of sequencers

The second generation of sequencers, (called NGS, for next-generation sequencing) appeared around 2000 and relies on a two-step process. Firstly, the DNA to be sequenced is replicated multiple times using chemical reactions. Secondly, molecules named "polymerase" will "scan" each copy of DNA and emit light. The sequence of bases can be determined by measuring the light polymerase molecules emits, hence the method's name, "pyrosequencing". The main advantage of NGS over the first generation is that NGS allows parallel sequencing, dramatically decreasing the cost of each sequence and making it viable to sequence millions of bases [2]. These methods produce a large amount of read (millions of them). The reads they produce are typically a few hundred bases long, which is an order of magnitude smaller than the reads of the third generation.

Third generation of sequencers

The third generation of sequencers (which appeared around 2010) gets rid of the need for DNA replication and allows to sequence strands of DNA directly. Two methods are widely used: the solution of Pacific Bioscience (hereafter "PacBio") and the one from Oxford Nanopore Technologies (hereafter "ONT"). PacBio's solution allows visualizing fluorescent lights burst [21] produced by the polymerase. ONT's solution, called "nanopore", consists in passing DNA in a small hole in a membrane [22], on which an electrical tension is applied. As each base flows through the hole, its ions create an electrical signal, which is measured to infer which base passed through the hole. The main advantage of those methods is the long reads they produce (typically thousands of bases long). Nanopores

2. The human genome consists of about 3 billion bases. Sequencing this genome at 100 bases per hour would take about 3000 years (or the use of multiple sequencers in parallel).

can be embedded in machines about the size of a flash drive and directly connected to a laptop, making them usable everywhere. Nanopore sequencers can output hundreds of thousands of bases per run, but their high error rates (about 1% of erroneous bases in reads [23], mainly insertions and deletions) make them less suitable for some biological applications: as such, they are often paired with NGS to take this flaw into account.

1.3 Downstream analyzes

As stated above, sequencers can output a set of thousands or millions of reads per run. The huge number of reads produced by a sequencer makes them unsuitable for human beings to analyze without the help of computers. Thus, computer programs were developed to deal with those sets of reads, performing various tasks, including reconstructing the original DNA sequences from the reads or locating differences between two sets of reads. The development of these tools is one of the domains of a science called bioinformatics.

Most bioinformatic tools are computationally heavy. They may require intensive CPU usage and lots of RAM or disk storage (potentially terabytes).

1.4 Amount of data

As of April 2023, there are already about 47.5 petabytes of reads available on the European Nucleotide Archive. The number of reads available is growing at an exponential rate [24] (see Fig. 1.3), faster than computational power (reads' data doubles in size every 35.5 months). Given the sheer number of reads, it is impossible to run computationally heavy software on all of them.

It is even impossible to select datasets that contain a specific DNA subsequence. If you want to know in which datasets a sequence (*e.g.* a short sequence, a gene, a genome) is likely to appear, it is intractable, as datasets cannot be retrieved based on their content.

This leads to an unfortunate situation: DNAs are sequenced, analyzed, and forgotten in hard drives since there is no method to query them among the huge number of sequences publicly available. Imagine being a biologist and stumbling across a fascinating DNA sequence. You may want to know whether this sequence was already known and placed in

a publicly available database. Unfortunately, there is currently no method for performing such a task on reads.

Providing a way to select datasets based on a queried sequence is the main focus of this work.

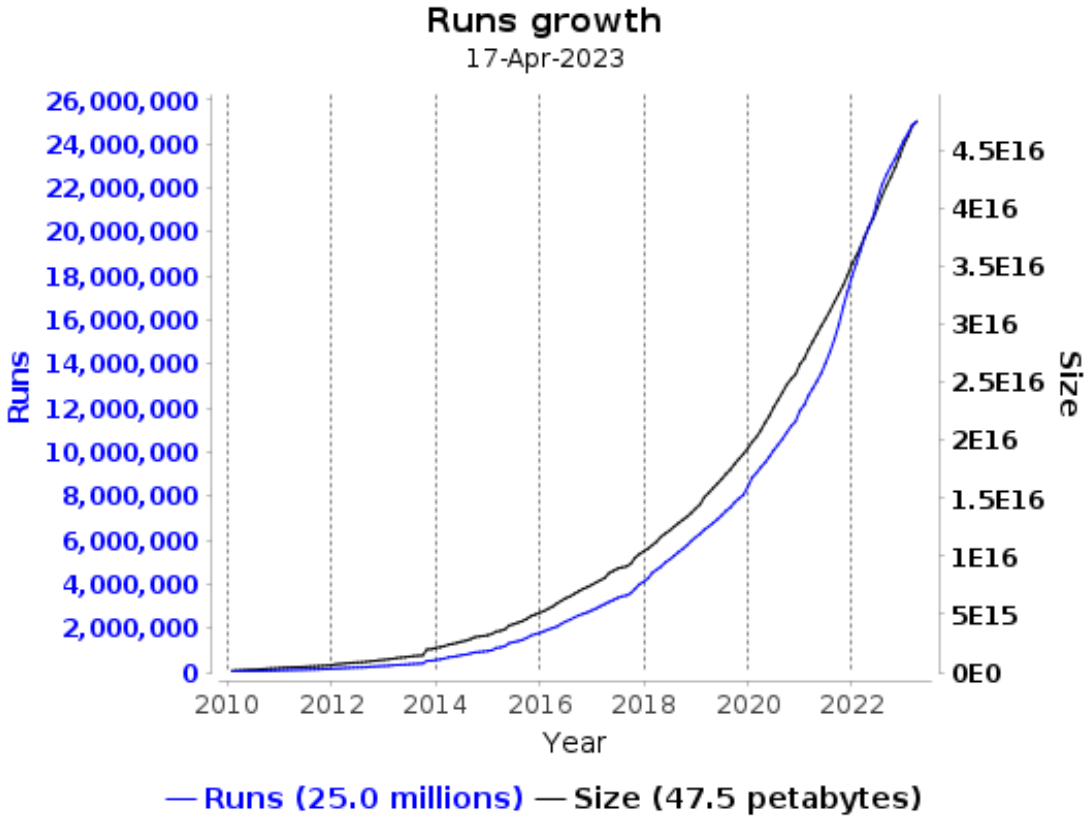


Figure 1.3: Data available on the European Nucleotide Archive up to April 2023. Figure from EBI.

1.5 Search engines

We call "dataset" a file containing a set of reads and "query" a sequence.

In the context of this work, a search engine is a tool that tackles the two following problems:

presence-absence Given a set of datasets and a query, identify in real-time every dataset

in which the query likely occurs.
counting Report the abundance of the query in the datasets in which it likely occurs.

The naive approach to answering these two problems is to compare the query with every read of every dataset. However, given the size of the datasets, this is impossible in practice: each query would take days, weeks, months, or even years, depending on the size and the number of datasets. [3]

That would be like searching for a piece of information on the internet by manually scrolling through every website: i.e. a considerable waste of time. Instead, internet users rely on search engines, that quickly and efficiently indicate where the data of interest are located. It is then up to the user to read the data to extract the piece of information, but this time on a much smaller scale (typically a few web pages).

Similarly, search engines on DNA aim to index sets of biological sequences (reads, genomes, ...) to later pick datasets in which a query is likely to occur. These methods rely on a preprocessing step, called indexation, performed on the datasets before actually answering any query.

Indexation

The goal of the indexation step is to compute a data structure which allows preventing the reading of the original datasets each time a query is submitted to the search engine. At indexation time, the set of datasets is turned into an intermediate representation called an **index**. This index is made so that queries against it are faster than on the original dataset. Typically, for a given query, the query time without an index grows linearly with the number of reads in each dataset, while query time against an index grows, in the worst case, linearly with the number of *dataset*.

The index is stored on external storage to avoid having to recompute it for each query. If the index is small enough, it can be stored directly in the main memory. If the index is stored on external storage, when a query is submitted to the search engine, the index is either loaded from the external storage into the main memory and queried, or directly queried from the external storage.

Importantly, a query on an index may, or may not, yield false positive results as we will explain in section 1.5. Intuitively, an index yielding false positive results may indicate

3. Note that the number of available datasets is increasing so quickly that once your query is finally done, new datasets would be available. Your query would be severely outdated before it is even finished!

that a subsequence of the query appears in a dataset, even if does not. False positives are one of the main drawbacks of some methods.

In this work, we tackle the challenge of lowering the number of these false positives.

When indexing a website composed of natural texts in a search engine, the strings to be indexed usually consist of a sequence of sentences. Each sentence has a beginning (an uppercase letter) and end (a full stop) and is made of words, which are drawn from a relatively small set (the number of words currently used in English is about 1.7×10^5). Each word belongs to a "part of speech" (name, verb, participle, article, pronoun, preposition, adverb, or conjunction).

Indexing and querying words is handy because words allow a partial match between sequences, which makes the search engine tolerant to mistakes, like typos in a text (or mistakes introduced by sequencers in the case of DNA sequences).

The challenge in indexing DNA reads is that they are not divided into sentences or words, since there is no space symbol in the DNA alphabet. Two reads may come from a similar DNA sequence, but be staggered relatively to one another; to compare them, one must align them beforehand, which is time-consuming.

Since dealing with sets of words allows handling the challenge of alignment, bioinformatic tools use a word-like concept: k-mers.

k-mers

Given an integer k, the k-mers of a sequence are all subsequences of consecutive k bases the sequence contains. See the left part of Fig. 1.4 for an example of the k-mers of a sequence. Given two sequences, a k-mer occurring in both sequences is called "shared". The number of shared k-mers between a queried sequence Q and a dataset provides an insight into the presence of Q in the dataset.

Constructing an index representing a dataset basically means computing k-mers from every read of that dataset, then adding those k-mers in the index. Querying such an index with a DNA sequence means computing k-mers from that sequence, then querying those k-mers against the index. If a k-mer is not found in the index, then it is absent from the dataset. Else, it is assumed to be present. Importantly, false positives generated by the index tend to overestimate the proportion of shared k-mers.

In the context of a search engine, a query is said "found" in a dataset if the proportion

Figure 1.4: k-mers from both the original read and its reverse complement. Observe that no k-mers from the leftmost read match any k-mers from the rightmost read (which represents the other DNA strand).

of shared k-mers between the query and the dataset is higher than a predefined threshold. Reducing the probability of occurrence of false positive k-mers is thus crucial to provide an accurate response to the end user.

The challenge tackled in this work is to build an index representing hundreds of billions of distinct k-mers from a dataset.

Ideally, since genetic information is the same on the two strands of a DNA molecule, we would like to find a perfect similarity when indexing reads from a DNA strand and querying the other strand. However, if we compare a read and its reverse complement via their k-mers content, it is unlikely to find a match between them. See Fig. 1.4 and observe that no k-mer from the read matches any k-mer from the other strand.

As such, comparisons are done through the use of the canonical form of k-mers. The canonical form of a k-mer is the minimum between a k-mer and its reverse complement (i.e. the same location, but sequenced from the other strand). In this work, the order of k-

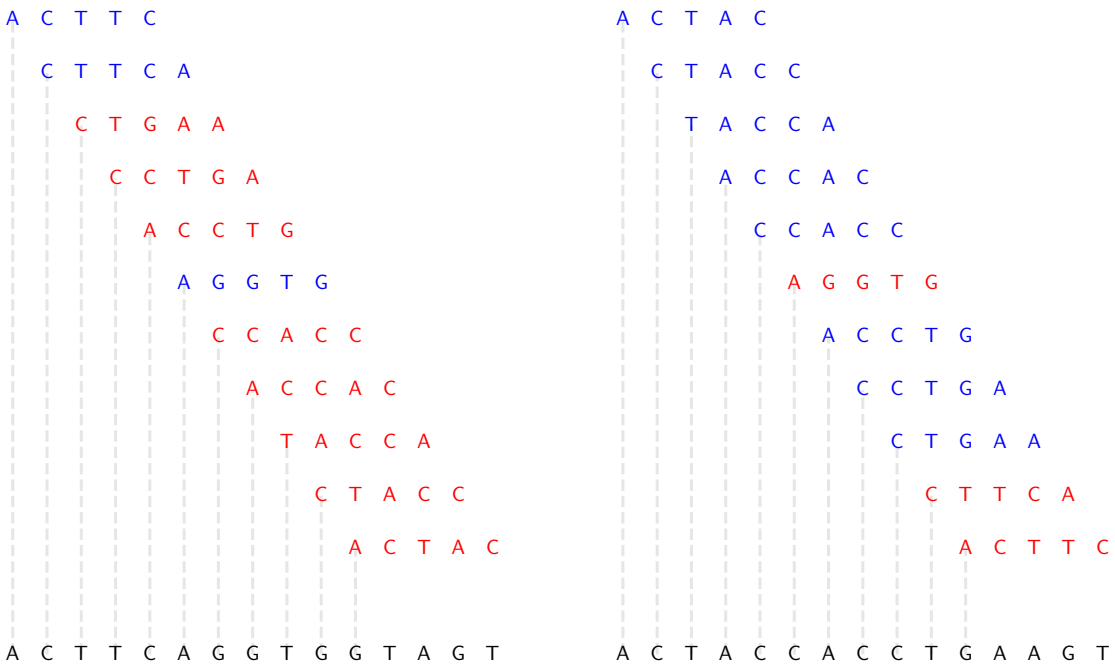


Figure 1.5: Canonical k-mers from both the original read ("ACTTCAGGTGGTAGT") and its reverse complement("ACTACCACCTGAAGT"). Note that in practice, a typical value of k is 31. Observe that canonical k-mers from the leftmost read match canonical k-mers from the rightmost read (which represents the other DNA strand). k-mers that were turned into their canonical form are represented in red.

mers is illustrated by the lexicographic order, although in practice k-mers are hashed and generally ordered by their hash values. Using canonical k-mers allows us to not care about which strand a read comes from. It allows retrieving the similarity between a sequence and its reverse complement (as illustrated in Fig. 1.5).

k-mers with an even number of bases can be their own reverse complement (*e.g.* "AATT"). In such case, the start of a k-mer would be equal to the reverse complement of its end. It would lead to a loop in the underlying De Bruijn graph (defined in the next chapter), which is not handled well in tools trying to guess the original sequence from its k-mers. With an odd number of bases, that is impossible, as the base in the middle of the k-mer would have to be its own reverse complement. Thus, bioinformaticians usually choose an odd length for k-mers.

If k-mers are too short, they have a high probability of appearing by chance in datasets (i.e. they have a low specificity). If k-mers are too large, they might be too specific to a read and not appear anywhere else. In this work, a typical value of k is 31, since it yields good practical results while being the maximal odd number of bases that can be encoded in 64 bits.

Space requirement

In this section, I will explain why the space requirement for indexing k-mers is important. To simplify, I will only consider the presence-absence setting and exclude the concept of canonicity.

Since each base is drawn from a four-letter alphabet $\Sigma = A, C, T, G$, the number of possible distinct 31-mers is $4^{31} > 10^{18}$. This is thirteen orders of magnitude greater than the number of words in a natural text, thus the need to use specific methods to deal with DNA reads.

Each dataset containing i distinct k-mers can be viewed as drawing i samples from the set of all possible k-mers. A data structure able to represent each set must thus consist of 4^k bits.

- *Proof:*

  ◊ A dataset corresponds to one of the $\binom{4^k}{i}$ possible combinations. Since a dataset can have between 0 and 4^k distinct k-mers, the total number of possible distinct k-mer set is $\sum_{i=0}^{4^k} \binom{4^k}{i} = 2^{4^k}$. To represent each of these k-mer sets in a data structure, at least $log_2(2^{4^k}) = 4^k$ bits are required.

One approach to represent a k-mer set in such a data structure is thus to use a boolean vector v of length 4^k. Each k-mer is mapped to a unique number from 0 to $4^k - 1$ using a bijective function f. In this representation, for each k-mer α in the set, the value at position $i = f(\alpha)$ in v would be set to *true* (represented as a bit set to 1 in Fig. 1.6). Every other position in v would be set to false. Checking if a k-mer β is in the set only requires checking the value at position $f(\beta)$ in v.

However, modern machines do not have enough memory to accommodate 4^{31} bits (which is 576 PB, or 512 PiB), which raises the need for a smaller data structure. This results either in **i/** a datastructure that is specialized to index some k-mer sets only (like exact methods, defined in the next chapter, Section 2.2.3), or **ii/** a datastructure made to

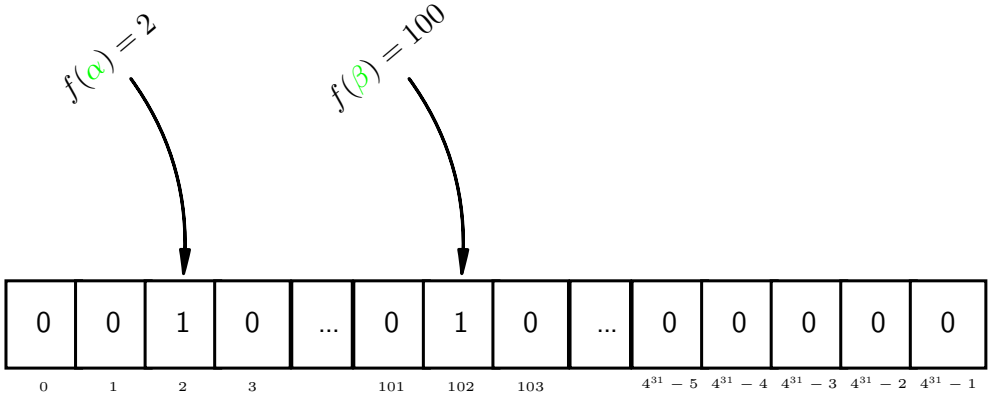


Figure 1.6: The datastructure (in black) can represent k-mers (in green) via the function f which associates each k-mer with a position in the vector. The size of the vector is 4^{31}, which is exactly the number of possible 31-mers.

index any k-mer set, but with a non-null false positive rate, as defined in Section 2.3. The concept of false positives arises from the fact that no data structure can have enough states to uniquely associate each state with a single dataset. Due to the pigeonhole principle, at least one state will represent multiple distinct datasets, making it impossible to identify which dataset is represented in the index with certainty.

For instance, one approach to reduce memory usage in the previously described data structure is to set a memory limit, denoted as m, and use a surjective function $g : k\text{-mers} \rightarrow [0; m-1]$ (with m $< 4^k$) to associate each k-mers to a corresponding cell in the boolean vector v (e.g. as $g(x) = f(x) \mod m$). Since the function is surjective, distinct k-mers can be associated with the same cell. This data structure is called a **Bloom filter** [25]. This way, distinct k-mers may be mapped to the same cell, resulting in ambiguity. Two k-mers such that $g(x) = g(y)$ while $x \neq y$ are said to be **colliding**. If

two k-mers x and y are colliding, then it is uncertain whether the cell contains information about x or y. Consequently, if the presence of x is stored in the vector while y is absent, querying for the presence of y will yield a false positive (see Fig. 1.7).

A way to reduce the probability of a false positive is to associate each k-mer with multiple hash functions (say, h hash functions per k-mer). Inserting a k-mer would require setting its h corresponding cells to $true$. At query time, a k-mer would be found if and only if its h cells are *all* set to $true$. To falsely report an absent k-mer as present, h collisions would be required, reducing the probability of a false positive rate. However, setting h cells per k-mer increases the number of cells set to $true$, increasing the probability of a collision. As such, depending on the number of elements in the filter ($\#elem$), there is a theoretical best value for h, which is $h = \lfloor \frac{m}{\#elem} \times ln(2) \rfloor$.

In practice, some methods, like kmtricks [4], require $h = 1$ for their inner working.

Indexing the abundance of k-mers can be done by using a very similar datastructure. Instead of using a vector of boolean values, recording the abundance relies on a vector of integers.

The way to deal with colliding k-mers having different abundances is to only record the maximum abundance value of these k-mers. This data structure is called a **counting Bloom filter** [26]. Importantly, note that querying for the value of a k-mer whose abundance was superseded by another colliding k-mer will then lead to an overestimation.

A datastructure recording presence of elements in a dataset while having a non-null false positive rate is called an Approximate Membership Query filter (`AMQ` filter for short). Similarly, a datastructure that indexes the count of elements in a dataset with a non-null overestimation rate is called a couting Approximate Membership Query filter (`cAMQ` filter for short).

The proportion of absent k-mers that would be falsely positive if queried against a filter is called the **false positive rate** of the filter. Similarly, the proportion of indexed k-mers that would be overestimated if queried against a filter is called the **overestimation rate** of the filter.

1.6 In this work

In this work, we do not propose a novel datastructure to index k-mers. We rather focus on improving all the existing search engines relying on an `AMQ` filter by decreasing their false positive rate and their query time. Given a memory budget, the method we propose

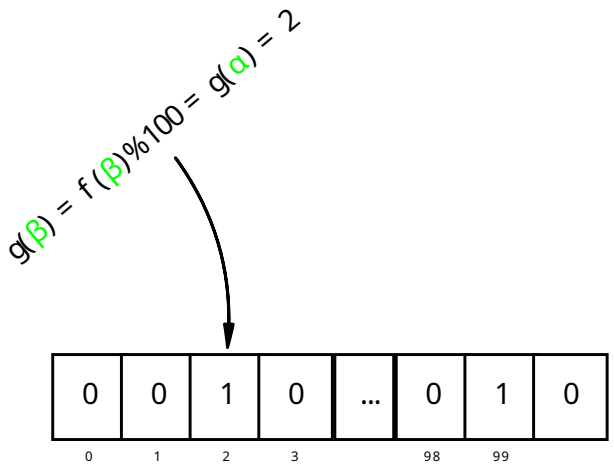


Figure 1.7: Example of a Bloom filter: the datastructure (in black) can represent 5-mers (in green) via the function g which maps each k-mer with a position in the vector. Note that it only has 100 cells (which is less than the number of cells required to represent each 5-mer), which may lead to a collision between two k-mers. Given this setup, it is impossible to record the presence of one colliding k-mer while not recording the presence of the other one as well.

may reduce the false positive rate of a Bloom filter from 5% to 0.056%, while introducing a threefold speed-up of its query time. Our method wraps around any `AMQ` filter (and does not require to modify it, keeping a maximal compatibility with existing methods) and works in different setups (indexing DNA or indexing natural texts).

We generalized our method to record the number of times each k-mer appears in a dataset using an `cAMQ` filter. We managed to lower the overestimation rate of the `cAMQ` filter as well as its false positive rate.

This work gave rise to two publications: a conference paper at SPIRE2021 [27] and another one in the Bioinformatics journal [28].

The method we propose was successfully applied on kmindex [6], a tool for querying k-mers, based on kmtricks [4] (a tool counting k-mers and constructing Bloom filters). We managed to reduce the false positive rate of its Bloom filters by orders of magnitude, allowing us to find a near-perfect similarity estimation between queries and datasets even when relying on an `AMQ` filter whose original false positive rate is 20%.

In Chapter 2, I introduce some notations and present the state of the art of k-mer

indexation. In Chapter 3, I will present our contribution, `findere`, which lowers the false positive rate of `AMQ` filters indexing k-mers. In Chapter 4, I describe another contribution, `fimpera`, which reduces the overestimation of a `cAMQ` filter indexing k-mers as well as their false positive rate. Finally, I will present my conclusions.

Chapter 2

State of the art of k-mers indexes

As stated in Section 1.4, providing a method to select datasets in which a queried sequence likely occurs is the main focus of this work. The selection of a dataset is based on the proportion of shared k-mers between the query and the dataset.

In the previous chapter, I stated that methods indexing k-mers rely on a preprocessing step, called indexation, that outputs a data structure recording the set of k-mers from a dataset. This data structure can either record the set of k-mers in an exact fashion or in an inexact way in a data structure called an Approximate Membership Query filter (`AMQ` filter for short).

Before diving into details, I will quickly recall the fundamental problem we are aiming to solve. Given a set of k-mers K and a query k-mer q, determine whether $q \in K$. In practice, we have multiple sets of k-mers ($[K_0, ..., K_N]$) and we want to determine all indices i such that $q \in K_i$. A more general problem is handling multisets of k-mers. In a multiset of k-mers, each k-mer is mapped to an integer, that we call **abundance** in our context. The problem is then to determine the abundance of q in each k-mer multiset $[K_0, ..., K_N]$.

A naive solution is to store each set of k-mers in a hashtable, which is available in most programming languages. However, generic hashtables are made to handle generic strings over the ASCII alphabet (while k-mers are overlapping words whose alphabet contains 4 letters). Storing overlapping k-mers in a generic hashtable would lead to the overlapping strings being duplicated across the overlapping k-mers. As such, generic hashtables are a sub-optimal solution to store k-mers. In practice, a single metagenomic seawater dataset represented as its set of k-mers in a generic hashtable can require about a hundred gigabits of memory. This memory requirement depends on the dataset. The required memory would become unpractical when dealing with multiple sets of k-mers.

Thus, methods were developed to efficiently and exactly represent large sets of k-mers. However, they do not handle well complex datasets, such as metagenomic datasets (metagenomic datasets are defined Section 2.2.3). For those datasets, most methods rely

on data structures called `AMQ` filters, which basically allow for a trade-off between their space usage and their false positive rate.

In the context of this work, a positive k-mer is a k-mer present in a dataset. A negative k-mer is a k-mer absent from a dataset. If a negative k-mer is found in an `AMQ` filter representing that dataset, it is a false positive. Otherwise, it is a true negative. A positive k-mer is always found in an `AMQ` filter representing that dataset.

The false positive rate of an `AMQ` filter is defined as $FPR_{AMQ} = \frac{\#FP}{\#FP+\#TN}$, with $\#FP$ denoting the number of false positive k-mers and $\#TN$ the number of true negative ones. The false positive rate of an `AMQ` filter mainly depends on its size.

In this work, we focus on indexing any dataset, including metagenomic datasets. As such, we will use an `AMQ` filter. However, our contribution does not focus on proposing yet another `AMQ` filter but rather focuses on **improving** existing `AMQ` filters (typically, by lowering their false positive rate for a given size). To the best of our knowledge, very few methodological developments have been done on improving existing `AMQ` filters. In fact, the only work in the same spirit is `kBF` [29].

As such, understanding the state of the art of exact methods indexing k-mers is not mandatory to understand our contributions. It is not even mandatory to know any `AMQ` filter to jump to the contribution part of this manuscript. However, understanding the state-of-the-art helps understand *why* our contribution is useful, while the next chapters will present *how* we contributed.

In this chapter, I will describe the landscape of methods that represent sets of k-mers. I will start by describing a concept that is widely used in bioinformatic tools, `minimizer`. I will then describe methods that represent sets of k-mers in an exact way (introducing the concepts of the De Bruijn graphs, `spectrum preserving string sets` and `minimal perfect hash function`), and then I will focus on methods indexing k-mers set in an inexact way (using `AMQ` filters). Finally, I will present `kBF`, a method aiming to improve existing inexact methods.

2.1 Basic concepts

2.1.1 Definitions and notations

A string S of length $|S|$ is a sequence $S[0]$, ..., $S[n-1]$ of characters from a finite alphabet. Given a string s, we define two k-mers as **neighbors** in that string if they are start in s at two consecutive positions, e.g. the 3-mers "AAC" and "ACT" are neighbors in "'GGAACTGG".
The notion of set can be extended into the notion of multiset. Unlike a set, a multiset can hold multiple instances of the same element, i.e. the multiset {0,0,1} is valid. The number of times an element appears in a multiset is called the multiplicity (e.g. the multiplicity of 0 in {**0**,**0**,1} is two). Traditionally, the cardinality of a multiset is the sum of the multiplicity of its elements (e.g. the cardinality of {0,0,1} is 3). However, in our context, the size of a multiset is defined as the number of distinct elements it contains.

2.1.2 Minimizers and super-k-mers

In this section, I introduce the concept of minimizer. Minimizers are the building block of some bioinformatics concepts, as in the definition of `monotigs` and the indexation tool "BLight".

Given an integer $m, m < k$, the **minimizer** [30] of size m of a k-mer is the smallest m-mer of the k-mer (see Fig. 2.1). In my examples, I will illustrate the notion of smallest m-mer with the lexicographic order (in practice, the ordering of m-mers may differ). Two overlapping k-mers are likely to share a minimizer, thus grouping in memory k-mers by their minimizer is a simple way to store overlapping k-mers close to each other in the memory. This helps reducing the number of cache misses when accessing a sequence of overlapping k-mer, which typically happens when querying k-mers extracted from a string.

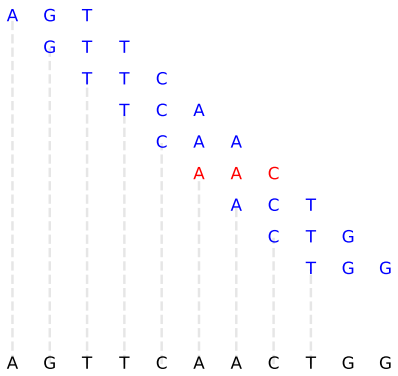


Figure 2.1: The 3-mers of the 11-mer "AGTTCAACTGG" are shown in blue and red. The minimum of the 3-mers, the minimizer, is shown in red, while the other 3-mers are represented in blue.

In a sequence, merging all consecutive k-mers having the same minimizer creates a super-k-mer [31]. For instance, the 11-mers of the sequence "TTAAAGTTCAACTGG" are:

- "TT**AAA**GTTCAA"
- "T**AAA**GTTCAAC"
- "**AAA**GTTCAACT"
- "AAGTTC**AAC**TG"
- "AGTTC**AAC**TGG"

Their minimizer of size 3 is shown in bold font. As such, its super-k-mer are "TT**AAA**GTTCAACT" and "AGTTC**AAC**TGG". Note that these 5 11-mers span only two minimizers, illustrating the fact that two neighbours k-mers are likely to share a minimizer.

2.2 Exact representations of k-mer sets

Some exact representations of a k-mer set K rely on two components:

- a function that maps each k-mer in K to a unique integer ($\in [0; |K|[$), called its identifier (while returning a random identifier for non-indexed k-mers);
- a datastructure storing K in a compact way.

Checking if a k-mer is in K is then a two-step process: first, get the identifier of a k-mer, and then verify if the k-mer is actually in K using the identifier and the compact representation. This verification is necessary because even if the queried k-mer is not in K, a valid identifier would be found. When indexing k-mers, the first step is done by using an `MPHF`, while the second one is done by using an `SPSS`. I will describe the two steps in the sections below.

2.2.1 Minimal perfect hash functions

An `minimal perfect hash function` (`MPHF`) is a function f that bijectively maps a set of k-mer K to a range of integers $[0; |K|-1]$. It could be implemented using a hashtable, that maps each k-mer in K to an integer, but that would require explicitly storing both the keys (the k-mers) and the values (the integers $\in [0; |K|-1]$). This would take $\mathcal{O}(|K| \times k)$ bits to store the k-mers and $\mathcal{O}(|K| \times log(|K|))$ bits to store the integers, leading to a solution of $\mathcal{O}(|K| \times (k + log(|K|)))$ bits. Note that an `MPHF` is allowed to return any integer for k-mers not in K (thus storing the keys is not mandatory). It is, in fact, possible to reach far less storage consumption, in $\mathcal{O}(|K|)$, using only $log_2(e) \approx 1.44$ bits/key.

`MPHFs`' important properties are their space consumption, their construction time and their lookup time. In practice, there exists a tradeoff between those properties. In the genomic sequences context, there exist multiple methods for creating an `MPHF` on k-mers.

A method for constructing an `MPHF` is BBHash [32]. BBHash requires multiple disk accesses per k-mer query. A parameter allows tuning the number of disk accesses. Reducing the number of disk accesses reduces the query time, but doing so increases its required disk space. Computed on one billion keys, this parameter allows reducing the query time from 271 ns to 179 ns per k-mer, but at the cost of increasing the memory usage from 3.1 to 6.9 bits per k-mer. To date, BBHash is one of the most rapid `MPHF` to build.

Another method of an MPHF is PTHash [33], which focuses on the query time, at the cost of a higher construction time. It is, indeed, about 4 times quicker to query compared to BBHash, but its construction time is about 3 times higher.

Both BBHash and PTHash methods are the best general purpose MPHF for k-mers, regarding the construction time and the query time respectively.

However, very recently, a *specialized* MPHF was proposed to beat the theoretical bound of 1.44 bits/key. LPHash [34] uses the concepts of SPSS (defined in Section 2.2.2) and super-k-mer to reduce its space usage. This is achieved by exploiting the fact that consecutive k-mers share $k-1$ letters, which is not exploited by other general purpose MPHFs. Very interestingly, LPHash initiates the study of MPHFs in the specific context of k-mers indexation. LPHash's query time is also lower than PTHash, one of the most rapid general purpose MPHF to date.

Importantly, this specificity of k-mer sets (the overlapping of elements) has been previously used to store the keys themselves. This gave the concept of SPSS, which is described in the following section. By storing the keys, an SPSS allows detecting absent k-mers; as such, it can complement an MPHF, which by itself is unable to do so.

Note that LPHash's method assumes that the k-mers are extracted from long sequences, which limits what they call the "fragmentation". As such, small reads, or complex metagenomic datasets are not their use case.

2.2.2 De Bruijn graph and SPSS

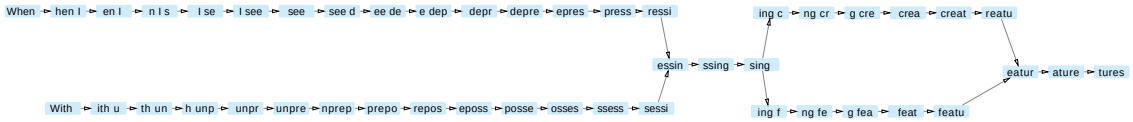
An SPSS [35] of a k-mer set K is a set of strings that contains every k-mer in K (and only those k-mers).

In this section, I will explain what a De Bruijn graph is and how to construct some SPSS from that graph.

De Bruijn graph

A de Bruijn graph [36] is an oriented graph in which nodes are k-mers. There is an edge between two k-mers x and y, from x to y, if and only if the $k-1$ suffix of x is the $k-1$ prefix of y.

In a de Bruijn graph, the edges can be computed from the nodes (it is possible to compute every 4 possible successors of a node and check whether they are on the graph or not). As such, any method that represents a set of k-mers also represents the



`de Bruijn graph` with $k = 5$

Figure 2.2: Taking two strings ("When I see depressing creatures" and "With unprepossessing features"), extracting 5-mers would lead to the following graph. Note that ambiguity is introduced: it is impossible to guess from the graph if "When I see depressing features" was in the original strings. Arrows are represented for clarity, but note that nodes carry enough information to compute edges on the fly. There are 46 nodes in the graph, each made of 5 letters, so there are $46 \times 5 = 230$ letters in the graph.

corresponding de Bruijn Graph. An example of a `de Bruijn graph` is shown in Fig. 2.2.

`unitigs`

In a directed graph, a *maximal non-branching path* is a path in which each intermediate node (thus excluding the first and the last) has exactly one predecessor and one successor. The first node of a non-branching path has 0 or multiple predecessors, and exactly one successor, while the last node has exactly one predecessor, and 0 or multiple successors.

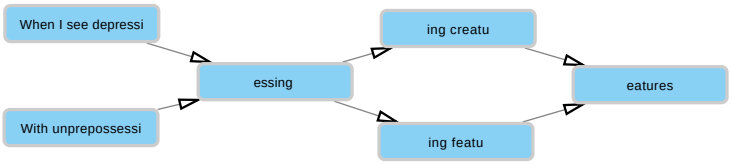
Observe that a `de Bruijn graph` contains duplicated information: by definition, two k-mers linked by an edge contain $k-1$ letters in common, that are duplicated across the two nodes.

To prevent this duplication, nodes on maximal non-branching paths are "merged" together without introducing ambiguity. When all nodes on all maximal non-branching paths have been merged, the nodes of the graph are called `unitigs`. The graph in which nodes are `unitigs` is called a **compacted de Bruijn graph**. See Fig. 2.3 for an example of such a graph.

Note that, by construction, a k-mer appears only in a single node of a compacted `de Bruijn graph`.

A compacted `de Bruijn graph` removes some repetitions of letters from the `de Bruijn graph`, but the original set of k-mers can still be recomputed from the set of `unitigs`.

Since `unitigs` contains all the original k-mers and only those k-mers, the set of `unitigs` is an `SPSS`.



compacted `de Bruijn graph` with $k = 5$

Figure 2.3: Taking two strings ("When I see depressing creatures" and "With unprepossessing features"), extracting `unitigs` would lead to the following graph. The graph contains 70 letters, which is less than the 230 letters of the `de Bruijn graph` shown in Fig. 2.2.

`unitigs` allow compacting the data, as illustrated in the following. Say we sequence reads from the two strings "When I see depressing creatures" and "With unprepossessing features". We may obtain the following set of reads: "When I see", "I see depressing c", "epressing creatures", "With unpre", "ng features", "unprepossessing featur" and "th unprepossessing feat". The total number of bases in all these reads is 113.
Using 5-mers, the number of bases in the `de Bruijn graph` computed from the reads is 230, but `unitigs` represent the same k-mer set using 70 bases.

`simplitigs`

The definition of `simplitigs` [37] is very similar to the one of `unitigs` but allows merging k-mers in a branching path (*i.e.* a path in which intermediate nodes may have more than one predecessor or successor). This allows merging more k-mers compared to `unitigs`. One way to think of it is that `simplitigs` allow merging k-mers across `unitigs`.

Applied in the example from the previous sections, a `simplitig` set could be "With unprepossessing creatures", "When I see depressi" and "ing featu". This set of strings can represent the k-mer set using 58 bases only. `simplitigs` are equivalent to another `SPSS`, called UST [35].

Note that the compaction of the k-mers is dependent on the topology of the underlying graph. If only a few k-mers are shared among the reads or if the sequenced genome is highly repetitive, then compactions remove less redundancy. This is typically the case when the

dataset contains reads from multiple distinct species, *i.e.* a metagenomic dataset and/or if the complexity in terms of k-mer representation is high.

The complexity of a dataset is a loosely defined term. A complex dataset usually has a high number of k-mers (a few billion), coming from repetitive genomes and various species.

Eulertigs

Eulertigs [38] propose to represent the `SPSS` of a set of k-mers using the minimal space theoretically possible. On a dataset containing 3682 E.Coli, a `unitig`-based representation needs 1.7 times more bases compared to using eulertigs, while UST requires 1.031 times more bases compared to eulertigs.

Thus, both UST and eulertigs can efficiently represent k-mer sets having a low complexity. However, computing eulertigs or UST is significantly more expensive in terms of memory usage compared to `simplitigs` (*e.g.* 79 GB for eulertig vs 9 GB for `unitigs` when indexing B. mori reads).

`monotigs`

In the context of indexing multiple multisets of k-mers, a `monotig` [39] is a merge of overlapping k-mers having the same `minimizer`, just like a super-k-mer. The difference with super-k-mers is that all k-mers in a `monotig` must have the same count profile across all k-mer multisets. If a k-mer in a `monotig` appears only in, say, multisets K_0, K_1 and K_5 with an abundance of 15, 3 and 4, then all other k-mers in that `monotig` must appear in those multisets (and in those multisets only) with the same abundances.

Conclusion on `SPSS`

The concept of `SPSS` spans multiple solutions reducing the number of bases required to represent a set of k-mer extracted from a biological dataset. The efficiency of these solutions relies on the topology of the underlying `de Bruijn graph`: in practice, they are not suitable for some applications, like compacting metagenomic datasets. As such, papers presenting `SPSS` usually evaluate their methods on multiple genomes from the same species and do not target metagenomic datasets.

However, searching for a specific k-mer in an `SPSS` requires scanning all the strings (*e.g.* `unitigs`, `simplitigs`) in the `SPSS`, resulting in a query time growing linearly with

the number of elements in the SPSS. This search can significantly be improved by the use of an MPHF, as described in Section 2.2.3.

2.2.3 Exact indexes of k-mers

One common way to build an exact index of k-mer extracted from reads is to use a combination of MPHF and SPSS. I will not describe every method of building an exact index, since this is not the focus of this work. Instead, readers can refer to the comprehensive surveys [40] and [41] for a broader description of k-mer indexes, the latter also including descriptions of inexact indexes and SPSS.

Bringing together MPHF and SPSS

Now that I have described the concept of a function that maps k-mers from a set K to a unique identifier and how to compact K using the concept of the SPSS, we have all the tools to perfectly index k-mers.

The key idea to index a k-mer set K is to compact K by computing an SPSS and to associate each k-mer in the SPSS with a unique identifier. At query time, the MPHF is used to retrieve the identifier that would have been associated with the k-mer if the k-mer was indexed. Using that identifier, the string (*e.g.* unitig or simplitig) the k-mer belongs to is retrieved. Then, if the k-mer is in that string, it is in the SPSS (and thus in K). Else, the identifier is incorrect, so the MPHF gave a wrong output, thus the k-mer is returned as absent from K. With an MPHF, **Pufferfish** [42] maps each k-mer to its position in the array of concatenated unitigs. Checking the position indicated by the MPHF prevents false positives.

BLight [43] is a k-mer hash table that is built from unitigs. At indexation time, unitigs are computed from the set of k-mers to be indexed. unitigs are then split into super-k-mers that are grouped by their minimizer. In BLight, a group of super-k-mers having the same minimizer is called a *bucket*. Then, for each bucket, super-k-mers are concatenated. In each bucket, an MPHF maps k-mers to an identifier. This identifier is associated with the rank of the super-k-mers they belong to.
When querying a k-mer, its minimizer indicates which bucket it belongs to. Then, the MPHF of that bucket is queried and returns the identifier of the k-mer. This identifier is used to retrieve the rank of the super-k-mer the k-mer belongs to. This rank allows retrieving the super-k-mer. Then, if the queried k-mer belongs to the super-k-mer, it is

present in K. Otherwise, it is not.
BLight is also compatible with other (potentially more compact) `SPSS`, allowing it to reduce its required disk space compared to using `unitigs`.

Bifrost [44] is a method for constructing, indexing and querying the compacted `de Bruijn graph`. In the context of indexing multiple k-mer sets, Bifrost supports recording in which set a given k-mer appears. In Bifrost, `unitigs` are stored in an array. Using a hashtable, `minimizers` are mapped to tuples made of **1/** the identifier of the `unitigs` they appear in and **2/** the starting position of that `minimizer` in that `unitig`. Given N k-mer sets, each `unitig` u of size $|u|$ is associated with a binary matrix BM of size $|u| \times N$ (this association is done with an `MPHF`). The i^{th} k-mer of u appears in the j^{th} set if and only if $BM[i][j]$ is set to *true*. A major proposal of Bifrost is to compute the `SPSS` on the fly, using Bloom filters to approximate the `de Bruijn graph` as a first step and to detect and remove k-mers present only once in a set. This approximated graph is then cleaned by iterating on the set of reads, allowing to detect false positives k-mers. This allows Bifrost to not use any external memory during the construction of the compacted `de Bruijn graph`.

SSHash [45] is built on top of PTHash [33] and super-k-mers. However, unlike BLight, SSHash does not split strings into super-k-mers but rather indexes the starting position of each super-k-mer in the strings. Not breaking strings into super-k-mers allows for preventing duplicating bases across two overlapping super-k-mer, sparing disk space.

Note that k-mers extracted from a sequence share, by design, $k-1$ overlapping bases with their neighbors. When querying a sequence, one will thus query a sequence of overlapping k-mers. This setup is called a streaming query. As streaming queries are usually the typical use case of an index, optimizing the implementation for this specific use case is of high importance.

Both BLight and SSHash are optimized for streaming membership queries.

Metagraph

Metagenomic reads are reads sequenced *directly from the environment* (say, a glass of ocean water). As such, metagenomic datasets may contain a lot (potentially thousands) of different, diverse species. The higher the number of diverse species in the dataset, the more complex the dataset. To the best of our knowledge, only one exact method, **MetaGraph** [46], targets some metagenomic datasets (though it is not its main target).

MetaGraph is less efficient on those datasets compared to other types of datasets. For instance, MetaGraph can represent a human read dataset using $> 8 \times 10^4$ times less space than the original dataset. However, MetaGraph's index takes more space than the original dataset for some complex datasets. For instance, on a catalog of human gut reference genome (UHGG [47]) and RefSeq [48], MetaGraph requires 7.69 times and 1.61 times more space compared to the input gzipped data, respectively. As such, it is impracticable to run MetaGraph on *Tara* metagenomic datasets [5] for instance. For terabytes-sized input datasets, MetaGraph requires running on a cloud of machines, making it impossible to run on a single machine. This is one of its drawbacks, as running it is not feasible for many labs.

2.2.4 Adding count information

As BLight is compatible with any `SPSS`, it is also compatible with `monotigs`. Since `monotigs` have the same count profile across all k-mer sets, storing one count profile per `monotig` is sufficient to index the count of all those the k-mers in that `monotig`. This is exactly what is done in **REINDEER** [39]. In practice, REINDEER assumes that the counts' profiles of k-mers in a `unitig` are identical. This assumption allows getting rid of biases introduced by sequencers at the cost of introducing an approximation of the counts. Assuming that the counts' profile is identical in a `unitig` also introduces redundancy, allowing better compression of the counts.

Gazelle [49] also records the counts of k-mers across multiple multisets, but only stores the log of counts. This transformation allows for saving space by clustering together different count values, at the price of losing precision in the count information.

2.2.5 Conclusion on exact methods

While exact methods are effective to index some k-mer sets, their effectiveness depends on the topology of the underlying `de Bruijn graph`. As such, exact methods are very effective to index a single DNA string (which was beforehand reconstructed from the reads, which is computationally costly). They are also effective in indexing reads from a single species (or a set of similar species), like reads sequenced from bacteria (as demonstrated in SSHash [45]). That is because the underlying `de Bruijn graph` consists of long non-branching paths (`unitigs`) if the species share long common sequences of DNA. As such, most of the exact methods do not target indexing complex metagenomic datasets.

Furthermore, most of the methods are static; once an index is computed, it is not possible to add another k-mer to that index (as it would require recomputing the `MPHF`).

2.3 Approximate methods for indexing sets of k-mers

As stated above, exact methods do not target indexing large and complex metagenomic datasets. Inexact methods are handy to index these datasets, as their efficiency does not depend on the topology of their underlying `de Bruijn graph` (only on their size, but not on the proportion of overlapping of k-mers). Those methods are relying on the use of Approximate Membership Query filters (`AMQ` filters for short).

An `AMQ` filter can conceptually be represented as a "black box" that can index any set of elements in a space-efficient manner but with two major downsides:
a/ the only two possible interactions with an `AMQ` filter is **indexing an element** and **querying an element**. Queries are questions in the form of "Is element x indexed in the black box ?"; iterating on indexed elements is not possible.
b/ for elements that are *not* indexed in the `AMQ` filter, the `AMQ` filter might falsely report that element as present.

Definitions
An element found in an `AMQ` filter is said *present* in that filter.
If it is present in the set of indexed elements, it is a *true positive*.
Else, it is a *false positive*.
An element found in an `AMQ` filter is said *absent* in that filter.
If it is absent in the set of indexed elements, it is a *true negative*.
Else, it is a *false negative*. False negatives are impossible in an `AMQ` filter.

There exists a tradeoff between the false positive rate and the space usage of an `AMQ` filter: for a given `AMQ` filter method, the smaller the datastructure, the more false positive it yields [50].

In this section, I will start by presenting the most commonly used `AMQ` filters, then I will present some methods relying on them.

2.3.1 `AMQ` filters

Bloom filters, counting Bloom filter and Count Min Sketch

As stated in Section 1.5, a Bloom filter [25] is a datastructure that stores the presence of elements in a vector of boolean values v, all initialized at $false$. The location of the boolean value that corresponds to an element x is given by the hash of that element, say i. At indexation time, the location of the inserted element in the vector is set to $true$, *i.e.* $v[i] \leftarrow true$. At query time, if the location of the queried element is equal to $true$, then **maybe** that element was indexed (or maybe another distinct element was indexed to that position due to a hash collision).

Similarly, a counting Bloom filter [26] (`cBF` for short) uses a hash function to determine where to store the count of an indexed element, but store information in a vector of integers (all initially set to 0). At indexation time, when inserting an element of hash i and of abundance ab, its cell is set to the maximum between the content of that cell ab, *i.e.* $v[i] \leftarrow max(v[i], ab)$. At query time, the abundance of a queried element of hash j is at most $v[j]$, since $v[j]$ can be an overestimation of the abundance of the queried element due to a collision of hash value.

To reduce the probability of hash collisions, it is possible to use multiple hash functions per element. At indexation time, the locations indicated by *all* the hash functions are all set to $true$. In the case of a counting Bloom filter, each cell pointed by a hash function is set to the maximum between its content and the abundance of the indeed element. At query time, in a Bloom filter, the element is said "found" if and only if *all* its hashes point to locations set to $true$. In a `cBF`, the minimum of the values pointed by the hashes is returned.

Note that using h multiple hashes per element slows down the queries h times compared to using a single hash function. That is because h more computations are required at query time per query. The optimal number of hash functions, to reduce the false positive rate of a Bloom filter is $h_{opti} = -log_2(FPR_{AMQ})$. For $FPR_{AMQ} = 0.001$, the optimal number of hash functions is 10, slowing down the query time by about one order of magnitude compared to using only one hash function.

Furthermore, some tools, like kmtricks [4], are incompatible with the use of multiple hash functions.

A Count Min Sketch [51] (CM Sketch for short) is similar to h counting Bloom Filters of m bits, stacked on a matrix of size $h \times m$. A CM sketch has h independent hash

functions (one per row of the matrix). Each indexed element is hashed h times, and its count is stored in the i-th row in the position indicated by the i-th hash function (just like h independent counting Bloom filters). When querying a k-mer, the minimum of the values pointed by each hash function in each filter is returned. It is typically designed to use a sublinear amount of space with regard to the number of indexed elements.

(Counting) Quotient filters

Another way of storing elements in a vector is to split the hash value of each indexed element in two: the first q bits of the hash is called the *quotient*, while the second part is called the *remainder*, as done in the quotient filter [52]. The vector v is a vector of remainders. When inserting an element, its quotient gives a position in the vector v, at which position the remainder is placed if that position is not already occupied. Distinct elements sharing the same quotient are stored in consecutive locations, called runs. This may involve moving previously indexed elements when inserting an element in between a run. Three metadata bits are added to keep track of the runs, allowing to retrieve at query time which remainder corresponds to which quotient.

At query time, the quotient of the queried element indicates at which position to look for its remainder. The vector v is then scanned backward from that position to look for the start of the run spanning over this position. Then, it is scanned forward to check if the remainder occurs in the run corresponding to the quotient. The scan stops when the scanned elements correspond to another quotient, indicated by the metadata bits.

This allows tuning the size (*i.e.* the number of cells) of the vector v, depending on the quotient size. Indeed, the number of cells in v is set to 2^q. However, the smaller the quotient size q, the higher the chance of quotient collision, forcing to insert elements farther away from their original position (and thus requiring a longer scan at query time). This also increases the indexation time.

A Counting Quotient filter [53] works similarly. If a k-mer appears twice, then its remainder is stored in the vector as usual, but the next position in the vector will contain its abundance. To prevent confusing an abundance with a valid remainder, the remainders are stored in increasing order, and abundances are stored in a way that breaks that increasing sequence, allowing detection of the starting position of an abundance. The end of an abundance is indicated by repeating its associated remainder.

Other recent developments

Since building an AMQ filter is not the focus of this work, I will not describe every existing AMQ filter and their inner working. There are, indeed, a lot of works in this area, including:

- Cuckoo filters [54]
- xor filters [55] (that claims to be faster and smaller than Cuckoo filters)
- Binary Fuse Filters [56] (by the authors of the xor filter, claiming to be smaller than xor filters)

All these AMQ filters expose a similar interface: $store(element)$ and $is\_present(element)$. This interface is extended by adding $how\_much(element)$ in the case of cAMQ filters.

Remark a very interesting feature of Bloom filters: their inner vector does not depend on the order of insertion of elements. Furthermore, the position of an element in the filter does not depend on the presence of other indexed elements, nor on their order of insertion. As such, two Bloom filters indexing two similar k-mer sets will end up having similar inner states. As shown below, some methods rely on this property for compressing Bloom filters indexing similar datasets, or for querying a k-mer in multiple filters in a single memory access (instead of one per filter).

Furthermore, for applications using less than 7 bits per element, Bloom filters are a competitive solution [53]. Since using low memory for our index is one of our targets, Bloom filters are perfect candidates for this task.

2.3.2 Methods using AMQ filters

Since a single AMQ filter can only index a single dataset, a lot of recent methodological developments have been done to index multiple distinct datasets in AMQ filters in a space-efficient manner.

As stated above, two Bloom filters indexing similar sets of k-mers will end up being similar. Because having similar filters for similar datasets allows great compression ratios when compressing the filters, most methods described below rely on Bloom filters.

BIGSI and COBS

Indexing N datasets can be done by maintaining N distinct Bloom filters(one per dataset), each using a different hash function. However, a query would require computing N hash functions and checking each Bloom filter sequentially (introducing a lot of cache misses).

The idea behind BIGSI [57] is very simple: the N Bloom filters are constructed using the same size m and the same hash function. Then, the N filters are column-wise concatenated in a $m \times N$ matrix. This way, at query time, only one hash function is computed. Since the N cells corresponding to a hash value are stored close to each other in the memory, querying a k-mer gives the memory location of its N associated bits. This setup is called "bit transposed" [58], "bit sliced" or *interleaved* Bloom filters.

COBS [59] is similar to BIGSI, but allows using filters with different sizes, so that larger sets can be stored in larger filters. This is achieved by having filters of identical sizes grouped by *batches*. The position of an element in a Bloom filter is simply given by its hash value modulo the size of the filter. For each batch, the filters are interleaved. This removes the need to access multiple distinct memory locations when querying a k-mer in a bucket.

COBS is thus superseding BIGSI, which, according to the authors of COBS[1], is a prototype.

SBT and HowDeSBT

Remark that applying a bitwise OR on two Bloom filters A and B having the same size and same hash functions gives a new Bloom filter C. Interestingly, the filter C represents the union of the two sets indexed in A and B.

The main idea behind SBT is that the absence of a k-mer in C implies its absence in both A and B.

Instead of concatenating filters like BIGSI, SBT [60] constructs a binary tree of these filters. Each leaf in the tree is a Bloom filter, representing a set of k-mers. A node in the tree is a also filter, which represents the union of all its children. Each filter is added to the binary tree in a greedy fashion that clusters similar filters together, to minimize the size of the union set stored in nodes (and thus their false positive rate).

This way, at query time, if an element is not found in a node, the queries against its

1. which includes authors of BIGSI

child can safely be skipped, as it is certain that the element would not be found in the node's child. If a k-mer is found in a node, then its children must be queried, potentially down to the leaves. In the worst case, the element is present in all leaves, making it necessary to access every leaf.

Improving on SBT, HowDeSBT [61] prevents storing information when the two children A and B of a node C share the same state for a given position, *i.e.* whenever there exists i such that $A[i] = B[i]$. Each HowDeSBT node C conceptually consists of two boolean vectors, How and Det, initialized at $false$. Whenever $A[i] = B[i]$, $How[i]$ is set to that value. In this case, to indicate that all the nodes have the same bit set at that position, $Det[i]$ is set to $true$.

At query time, when searching for an element of hash value j, if at any node $How[j]$ is set to $true$, then the query can be skipped for the children. Indeed, it is guaranteed that all the children of that node have the value of $Det[j]$.

HowDeSBT allows getting rid of duplicated information across leaves whenever the leaves are the children of the same node. In this case, duplicated information is stored only once at a higher node. Compared to stacking filters, like COBS, this allows a reduction of the space required to represent Bloom filters indexing similar datasets. However, the usefulness of HowDeSBT depends on how close the indexed datasets are to each other. Moreover, HowDeSBT slows down queries, compared to simply stacking Bloom filters on top of one another like COBS. It thus introduces a tradeoff between the size of the index and the query time.

In the tests we performed in our metagenomic context, the space gain of HowDeSBT is about a factor 2, while the query time increases by a factor of 1300, making it nonviable if we want to answer a query in real-time.

PAC

A recent method, PAC [62], allows removing the need of walking down from the root to the leaf introduced by SBT. In PAC, Bloom filters are interleaved and concatenated in a matrix, like BIGSI. To prevent unnecessarily reading bytes from the matrix, PAC stores two integers for each row: the position of the first $true$ position from left to right and symmetrically, the first $true$ position from right to left. These two integers give an interval of datasets in which a k-mer may occur. Datasets outside that interval are not read, leading to fewer disk accesses.

PAC allows adding another dataset in its index by simply adding a new Bloom filter in

the matrix and updating its interval. However, if the new Bloom filter consists of frequent *true* cells, PAC loses its interest. That is because, for most rows, the first *true* position would be the first one and thus the interval would indicate to read most datasets.

Needle

Needle [63] is a recent method that aims at retrieving the median abundance of queried k-mer by relying on the abundances of their `minimizers`. Interestingly, Needdle does not store the counts of k-mers in a counting Bloom filter, but rather cluster `minimizers` having the same range of abundance together in the same Bloom filter. Thus, Needle requires q Bloom filters to store q ranges of abundance.

At query time, to determine the abundance of a query, its `minimizers` are searched in each q filter. The query time thus grows linearly with the number of filters. In practice, q is small (about 15), mitigating this impact on Needle's query time. In fact, Needle's query time is lower than REINDEER's by multiple orders of magnitude.

Conclusion on methods based on `AMQ` and `cAMQ` filters

Indexation methods relying on `AMQ` filters propose a tradeoff between the index size (HowDeSBT), the construction time (PAC), the query time (Needle) and the possibility of updating the index (COBS, PAC).

However, they all suffer from a non-null false positive rate and/or overestimation rate. Bloom filters are the most space-efficient `AMQ` filters when using only a few bits per element (*i.e.* less than 7). For such value, their false positive rate is high: recommended false positive rate can be as high as 25%, like in BIGSI. To compensate for this high false positive rate, those methods introduce a parameter θ. The query is answered as negative if the fraction of found k-mers is less than θ. This way, a single false positive k-mer will not trigger a false positive query answer. However, the use of this parameter makes it impossible to query a single k-mer. More importantly, it becomes way less informative to return which k-mer from the query is possibly shared with a dataset.

As such, a method, named `kBF`, focuses on reducing the false positive rate of Bloom filters, potentially serving as a buffer between the Bloom filter and any method relying on it.

2.4 Methods focusing on lowering the false positive of existing filters

2.4.1 kBF

As stated in the previous sections, AMQs are widely used in bioinformatics methods and tools. Improving them would thus lead to an improvement in lots of existing methods. This is our main contribution in this work. To the best of our knowledge, only one other method, named kBF, focuses on improving the use of a specific AMQ (namely Bloom filters) in the context of bioinformatics.

As such, kBF is the only method comparable to our work.

kBF comes in two version; 1-kBF and 2-kBF.

An overview of the core principle of 1-kBF is as follows: when checking for the presence of a k-mer in a Bloom filter, if it is found in the filter, then every one of its possible left and right neighbors is queried as well. If none of its neighbors is found, then the k-mer is said to be a false positive (as k-mers indexed from a read have their left or right neighbors indexed as well). Thus, 1-kBF yields a false positive if the filter yields two consecutive false positives (which is less likely than a single false positive).

To further reduce the false positive rate, kBF can check for the presence of two neighbors (one predecessor in the de Bruijn graph and one successor). This strategy is the core of 2-kBF. Note that k-mers from the beginning (reps. the end) of reads are likely to miss a left (resp. right) neighbor, which would lead to a false negative when querying that k-mer. To prevent any false negative response in this setup, the first and last k-mers of every input reads are stored in a traditional hash table. Those k-mers are called potential edges k-mers. As reads can overlap, a potential edge k-mer can, in fact, have a neighbor in another read. Thus, after every read is indexed, the potential edges k-mers are scanned to check whether they actually have a neighbor. In such a case, they are removed from the hash table. The remaining k-mers are called edges k-mers.

In 2-kBF, when a k-mer x is queried, the presence of x is checked in the filter. If x is not found, x is declared absent. However, if x is found, it might be a false positive. As such, all of its potential left and right neighbors are computed and queried (by brute force searching for all its 8 possible neighbors). If the two neighbors are found, x is declared positive. If only one neighbor is found, the k-mer x is searched among the edges k-mers.

If it is not an edge k-mer, then it is declared absent. This avoids answering a k-mer as absent if it was indexed, preventing any false negatives.

The implementation of `kBF` is using a Bloom filter, but that method could be used on top of any existing `AMQ` filter.

One of the downsides of `kBF`'s method is the need to compute a hashtable containing edges k-mers, leading to a blow-up in memory usage. Moreover, since all potential left and right neighbors have to be computed at query time, this leads to $|\Sigma|+|\Sigma|$ additional queries for each k-mer (up to 9 queries per k-mer if $\Sigma = \{A, C, T, G\}$). The larger the alphabet, the higher the query time and the computational cost of queries.

Note that the `kBF` approach could be extended into checking the p potential successors of a k-mer (and not just the left (or right) neighbor). This would result in a more stringent definition of a positive k-mer, further decreasing the number of k-mers falsely reported as present. However, it implies computing the neighbors of each queried k-mer, leading to $|\Sigma| \times |\Sigma|$ queries just for the left of a k-mer. As such, the computational cost would be exponential with regard to p. Doing so would introduce a tradeoff between the computational cost of the query and the false positive rate. Moreover, the number of k-mer not having p successors would increase with p, leading to an increase in the memory usage of the hash table with regard to p.

As such, `kBF` is only suitable when applied to the DNA alphabet and checking only one left (and right) neighbor. The use of a hashtable makes `kBF` unsuitable in the context of small reads, as up to 6% of the indexed k-mers would be stored in the hashtable.

In our metagenomic context, compared to a Bloom filter, `kBF` reduces the false positive rate from 28.34% to 15.40% but slows down queries by a factor of 7.85 and increases memory usage. Due to the implementation of `kBF`, its memory consumption could not be measured. However, it required RAM usage two orders of magnitude higher than the underlying `AMQ` filter. As such, it is unsuitable.

2.5 Conclusion

Methods indexing sets of k-mers can be divided into two categories: exact and approximate methods. Most exact methods do not target indexing metagenomic datasets. One notable exception is Metagraph, which does not scale to our use case: complex metagenomic datasets. Approximate methods (*e.g.* COBS or PAC) have to deal with a non-null false positive rate, limiting their answer's precision, and preventing returning information at

the level of k-mers.

One method, `kBF`, focuses on reducing the false positive rate of Bloom filters. However, `kBF` requires storing a fraction of the input k-mers in a hashtable and introduces a tradeoff between the false positive rate and the query speed.

In the next section, I will present our first contribution, which leads to a decrease in the false positive rate of a filter, while *reducing* the query time and with *no* memory overhead. Our contribution allows reducing the false positive rate of a Bloom filter from 28.34% to as low as 0.95%, instead of `kBF`'s 15.40%, effectively proposing a 16-fold decrease compared to `kBF`. Furthermore, our contribution *reduces* the query time of the underlying `AMQ` filter instead of increasing it, like `kBF` does.

Chapter 3

First main contribution: `findere`

This chapter corresponds to our published paper in SPIRE 2021 [27]. Sections of the original paper were rewritten to add clarity and to match the structure of the manuscript.

3.1 Context

In the previous chapter, I stated that many methods indexing k-mers rely on Approximate Membership Query data structures, which are indexing k-mers with a non-null false positive rate. Reducing this false positive rate is of high interest, but to the best of our knowledge, only one method, `kBF` [29], focuses on tackling this issue. In this chapter, I will describe our contribution, which is a simple strategy for reducing the false-positive rate of any AMQ data structure indexing reads through their k-mers.

The method we propose, called `findere`, enables speeding up queries by a factor two and decreasing the false-positive rate by two orders of magnitudes. This achievement is done *without modifying the original indexing data structure*, without generating false-negative calls, and with no memory overhead. Alternatively, `findere` reduces the space required to represent an `AMQ` filter given a false-positive rate setpoint.

In our publication [27], we applied our method to Bloom filters. However, `findere` can be applied on any `AMQ` filter.

3.2 The `findere` strategy

The `findere` strategy comes from the following observations:

- If a k-mer is present in a string, then all its subwords are also present in the string.
- Conversely, if one of its subwords is absent in a string, a k-mer is absent from that string.

For instance, if a string contains the 8-mer "segfault", then the 6-mers "segfau", "egfaul", and "gfault" are present in that string as well. However, if a string does not contain the 6-mer "egfaul", then you can be sure that it does not contain the 8-mer "segfault".

`findere` relies on this principle. Instead of indexing k-mers in the `AMQ` filter with a false positive rate of $FPR_{AMQ} \in [0; 1]$, `findere` indexes smaller words (called s-mers, $k \geq s$) in the filter.

We note $z = k - s$ for the sake of brevity. A k-mer is thus composed of $z + 1$ s-mers.

In our publication, we used another set of notations. K-mers (from the original paper) correspond to k-mers in this manuscript, while k-mers in the original paper correspond to s-mers in this manuscript. Here, notations from the paper have been changed to be made consistent with the rest of the document.

At query time, `findere` reports a k-mer as "present" in an `AMQ` filter if and only if all its s-mers are found in it. This way, when querying an absent k-mer composed of absent s-mers, falsely reporting that k-mer as "present" would require falsely finding all its s-mers in the filter. In this case, the probability of finding all absent s-mers in the filter is ${FPR_{AMQ}}^{z+1}$, which is lower than FPR_{AMQ}. Given a classical FPR_{AMQ} value of 1%, with z = 3, the theorical probability to get three consecutive false-positives s-mers is 10^{-6}%. In practice, absent k-mers may be composed of present s-mers, so the false positive rate when applying `findere` on an `AMQ` filter, defined here as $FPR_{\texttt{findere}}$, is higher than ${FPR_{AMQ}}^{z+1}$.

Splitting k-mers into $z + 1$ s-mers and associating each s-mer with a hash value as done in `findere` may be seen as a similar strategy as using $h = z + 1$ hash functions per k-mer with a Bloom filter. There are three obvious observations to be made here:

- Canonical indexing methods such as HowDeSBT [61] or Cobs [59] for instance rely on Bloom filters using a unique hash function. These state-of-the-art approaches can thus directly benefit from our proposal, with no modification.

- Indexing and query time grow linearly with the number of independent hash functions used, while `findere` takes advantage of the s-mers shared between consecutive k-mers. Even if it is not restricted to this framework, `findere` is meant for querying streams of k-mers. In this situation, when streaming all successive k-mers from a sequence, a k-mer shares z s-mers with its predecessor. Only one unique new s-mer has to be inserted or queried for all k-mers except the first one. Thus, with `findere`, the computation time does not grow with the z value. Notably, as shown in Table 3.1, compared to a traditional `AMQ` filter, the query time even decreases when using `findere`, as once a s-mer is detected as absent, all k-mers that span this s-mer can be skipped.

- The number of hash functions of a Bloom filter has an optimal value (h_{opti}). Using more hash functions saturates the filter, leading to an increase in the false positive rate. In the `findere` approach, z is independent of h_{opti}.

At indexation time, the `findere` strategy simply requires indexing s-mers in the `AMQ` filter instead of k-mers.

The query algorithm is a bit more complex to explain than the indexing algorithm. I will thus explain the query by first describing a naive algorithm, before presenting two optimizations making the query faster to execute in practice. The query consists in finding which k-mers from a queried sequence Q have all their s-mers indexed in an `AMQ` filter.

A naive query strategy consists in querying each s-mers of each k-mers. Such implementation of `findere`'s query, in $\mathcal{O}(|Q| \times z)$, is given in Algorithm 1.

Algorithm 1 Naive `findere`'s query

1: **procedure** `FINDERE` _NAIVE_QUERY($Q \in \Sigma^*$; `AMQ` filter indexing s-mers; k and z in $\mathbb{N}^+$ with $|Q| \geq k, z < k$.)
2: $s \leftarrow k - z$
3: ▷ Index the presence-absence of all k-mers from Q
4: $presence \leftarrow booleanVectorOfSize(|Q| - k + 1)$ ▷ all values are initially set to *true*
5: **for** i in $[0; |Q| - k + 1]$ **do**
6: $kmer \leftarrow k$-mer starting at position i in Q
7: **for** $smer$ in $kmer$ **do**

```
            if smer ∉ AMQ filter then
                presence[i] = false
            end if
        end for
    end for
    return presence
end procedure
```

Note that, as mentioned earlier, neighbors k-mers share $z+1$ s-mers. Conversely, one s-mers is contained in $z+1$ k-mers. As such, a naive query requires checking if a given s-mer is present in the filter $z+1$ times.

I will now explain, in the next two sections, how using `findere` on top of an `AMQ` filter actually speeds up the queries on that filter.

3.2.1 Optimisation 1: prevent multiple queries per s-mer

We define a **positive stretch** of k-mers (resp. s-mers) as a consecutive sequence of positive k-mers (reps. s-mers). A positive stretch can consist of a single k-mer.

Observe that a positive stretch of p (with $p \geq 1$) k-mers implies a positive stretch of $p+z$ s-mers.

- *Proof by induction*:
  - ◊ *Base Case*: A positive k-mer ($p = 1$) implies $z+1 = z+p$ positive s-mers. Thus, the relation holds for $p = 1$.
  - ◊ *Induction Step*: suppose the relation holds for a given p. Adding a positive k-mer to a positive stretch of p k-mers implies adding a positive s-mer in the positive stretch of $p+z$ s-mers. Thus, if the relation is true for a given p, the relation holds for $p+1$.
  - ◊ *Conclusion*: The relation holds for any $p \geq 1$.

Similarly, a positive stretch of $p+z$ s-mers implies a positive stretch of p k-mers. Thus, an optimized query algorithm consists of identifying all the stretch of positive s-mers of size $p+z$ and reporting a positive stretch of p k-mers. This algorithm is given in Algorithm 2. This algorithm is in $\mathcal{O}(|Q|)$ time. Using it for `findere`'s query makes that

query time about the same as when querying k-mers on an AMQ filter; a small overhead is introduced, since this algorithm searches for s-mers instead of k-mers and there are z more s-mers in a query compared to k-mers, resulting on z more queries.

Algorithm 2 findere's query in $\mathcal{O}(|Q|)$

procedure FINDERE _QUERY($Q \in \Sigma^*$; AMQ filter indexing s-mers; k and z in $\mathbb{N}^+$ with $|Q| \geq k, z < k$.)
    $s \leftarrow k - z$
    $presence \leftarrow booleanVectorOfSize(|Q| - k + 1)$ ▷ all values are initially set to $false$
    $currentStretchLength \leftarrow 0$ ▷ length of stretch of s-mers
    for i in $[0; |Q| - k + 1]$ **do**
        $smer \leftarrow s$-mer starting at position i in Q
        if $smer \in$ AMQ filter **then**
            $currentStretchLength \leftarrow currentStretchLength + 1$
        else
            if $currentStretchLength > z$ **then**
                for j in $[i - currentStretchLength; i - 1]$ **do**
                    $presence[j] \leftarrow true$
                end for
                $currentStretchLength \leftarrow 0$
            end if
        end if
    end for
        ▷ last stretch
    if $currentStretchLength > z$ **then**
        for j in $[i - currentStretchLength; i - 1]$ **do**
            $presence[j] \leftarrow true$
        end for
    end if
    return $presence$
end procedure

3.2.2 Optimisation 2: prevent querying non-informational s-mer

In practice, using findere is *faster* than directly using the underlying AMQ filter. This section explains how we achieved this speedup.

As stated in 3.2.1, to search for positive k-mers, we only need to find positive stretches

of s-mers of size greater than z. As such, we are not interested in detecting smaller positive stretches of s-mers.

Suppose the s-mer starting at position i in a query is negative. If the s-mer starting at position $i + z$ in the query is negative as well, it implies that no positive stretch of more than z s-mers can start between position i and $i + z$. In such a case, the search for positive s-mers can jump from i to $i + z$ *without querying intermediate positions*. Thus, when stumbling across a negative k-mer, our optimization consists of "probing" only one out of z positions as long as the s-mers on these positions are negatives. This allows speeding up queries with regard to z.

We refer to that optimization as the "skip optimization". This optimization is implemented in Algorithm 3. Its effect is shown in Section 3.6.2.

Algorithm 3 `findere`'s optimized query

```
1: procedure FINDERE _QUERY_OPTIMIZED(Q ∈ Σ*; AMQ filter indexing s-mers; k and z in
   ℕ+ with |Q| ≥ k, z < k.)
2:     s ← k − z
3:     currentStretchLength ← 0
4:     extendingStretch ← true
5:     i ← 0
6:     while i ≤ |q| − k + 1 do
7:         smer ← s-mer starting position i in Q
8:         if smer ∈ AMQ filter then
9:             currentStretchLength ← currentStretchLength + 1
10:            if extendingStretch then
11:                i ← i + 1
12:            else
13:                extendingStretch ← true
14:                i ← i − z
15:            end if
16:        else
17:                ▷ last stretch
18:            if currentStretchLength ≥ z then
19:                print all overlapping k-mers occurring on Q from i − currentStretchLength
   to i − 1 − (k − s)
20:            end if
21:            currentStretchLength ← 0
22:            extendingStretch ← False
```

23: $i \leftarrow i + z + 1$
24: **end if**
25: **end while**
26: **if** *currentStretchLength* $\geq z$ **then**
27: print all overlapping k-mers occurring on Q from $|q| - k + 1 - currentStretchLength$ to $|Q| - k$
28: **end if**
29: **end procedure**

3.3 Limits of `findere`

3.3.1 "construction false-positives" (cFP)

Detecting k-mers based on their s-mer content leads to the apparition of a novel kind of false positive.

A negative k-mer may be composed of true positives s-mers. In such a case, `findere` will erroneously report that k-mer as positive. We name those false positive k-mers "construction false-positives" (cFP for short). A cFP is a false positive k-mer composed of only true positive s-mers (if one of its s-mer is a false positive, then the k-mer is not a cFP). Similarly to $FPR_{\texttt{findere}}$, which represents the proportion of false positive k-mers obtained when using `findere`, one can define $cFPR_{\texttt{findere}} = \frac{\#cFP}{\#cFP+\#TN}$ as the proportion of cFP among negative k-mers.

Example of a construction false positive: indexing the sequence 6-mers of the sequence "GGTCACTGACA" through their 3-mers requires indexing 3-mers, among them "ACT", "CTG" and "GAC". Querying the absent 6-mer "GACTG" would query the aforementioned 3-mers, which are indexed. Thus, "GACTG" would be found, even without any false positive 3-mer.

Since the occurrence of a cFP depends only on true-positive calls, it will be generated no matter the `AMQ` filter `findere` is applied to (even if the `AMQ` filter had a false-positive rate of 0). The smaller s, the higher the number of cFP, as when s gets smaller, random s-mers are more likely to occur by chance in the indexed dataset. Indeed, the probability of a random s-mers appearing by chance in a text is $1-(1-\frac{1}{|\Sigma|^s})^n$, with n being the number of characters. For instance, for $s = 11$ and a sequence as little as n =20 million characters, the probability of the presence of any 11-mer is $> 99\%$. For a fixed k, $cFPR_{\texttt{findere}}$ will

increase with z, since s will decrease.

In practice, when indexing billions of bases, using $s \geq 16$ is sufficient to keep $cFPR_{\texttt{findere}}$ orders of magnitude lower than $FPR_{\texttt{AMQ}}$. However, to ensure a low probability of yielding a cFP, we recommend using $s \geq 20$.

With the recommended parameter value ($z = 3$), the proportion of cFP is negligible compared to $FPR_{\texttt{AMQ}}$ when indexing 31-mers, as shown in Fig. 3.2.

`findere` is robust with regard to the z parameter. Using z from 3 to 14 if $k = 31$ yields good results, both in terms of query time and false positive rate, as shown in the Results section.

3.3.2 False positive at the extremity of a positive stretch

Another limit of the `findere` strategy is that the first negative k-mer (that we call "fragile k-mers") at the end of a true positive stretch of k-mers is more likely to be a false positive compared to a random negative k-mer.

On one hand, a fragile k-mer shares z true positive s-mers with its true positive neighbor. As such, only one of its s-mers is negative. Thus, a fragile k-mer only requires one false positive s-mer to be a false positive k-mer. This happens with a probability of FPR_{AMQ}.

On the other hand, as stated in Section 3.2, a random negative k-mer consists of $z+1$ negative s-mers.

Thus, not all negative k-mers have the same probability of being returned as a false positive by `findere`. Note that, in the worst case, the probability of falsely reporting a fragile k-mer as present is equal to the original FPR_{AMQ}.

3.4 Theoretical analysis

As stated in Section 3.3.1, the $cFPR_{\texttt{findere}}$ increases if s decreases. As shown in the results (Fig 3.2), the false positive rate of `findere` is increasing abruptly at a specific s value. To better understand this sharp increase in construction false positive rate, we worked on a theoretical analysis with a team of statisticians: Mahendra Mariadassou (Université Paris-Saclay, INRAE, MaIAGE, Jouy-en-Josas, France), Sophie Schbath (Université Paris-Saclay, INRAE, MaIAGE, Jouy-en-Josas, France), and Stephane Robin (MIA-Paris, Université Paris-Saclay, AgroParisTech, INRAE, Paris, France). Our objective was to find

which s values lead to a sharp increase in the false positive rate.

Together, we were able to generate an analysis of the probability of occurrence of construction false positives. This analysis is limited to random genomes: we used a model in which the probability of a base occurring in a genome is $\frac{1}{4}$ and independent of the neighbors of that base.

Consider:

- a random genome G of length $|G|$
- two word sizes k and s such that $s > k/2$
- a given word $W = (b_1, \ldots, b_k)$ (or $b_{1:k}$ in short) of size k

We want to compute the probabilities of the events:

- A: word W is present in G
- B: all $z+1$ s-mers of W, noted $w_1 = s_{1:k}, w_2 = b_{2:k+1}, \ldots, w_{z+1} = b_{z+1:k}$ are present in G
- $C = B|\bar{A}$: all s-length subwords of W are present knowing that W is absent from G (*i.e.* a construction false positive)

We call the left (resp. right) neighborhood of size z of $\bar{w}$ the z letters occurring before (resp. after) $\bar{w}$ in G.

Under the assumption $s > k/2$, the words $w_1, \ldots, w_{z+1}$ share the subword $\dot{w} = w_{z+1:k-z}$ of length $2s-k = k-2z$. Whenever $\dot{w}$ occurs, there is a possibility that some s-mers of W also occur. This possibility is completely determined by the left and right neighborhoods of size z of $\bar{w}$. Note $n_- = n_{-z:-1}$ the left neighborhood and $n_+ = n_{+1:+z}$ the right neighborhood.

For any word u, we note $\mu(u)$ the occurrence probability of u at any given position and $N(u)$ the number of occurrences of u in G.
We have;
$N(u) \sim \mathcal{B}(|G|, \mu(u))$
Since in practice $\mu(u) << |G|$:
$N(u) \sim \mathcal{P}(|G|\mu(u))$

Using the Poisson approximation for $P(A)$ is given by:

$$P(A) = P(N(W) \geq 1) = 1 - P(N(W) = 0) = 1 - e^{-|G|\mu(W)} \tag{3.1}$$

While $P(B)$ is given by:

$$P(B) = P(N(w_1) \geq 1; \ldots; N(w_{z+1}) \geq 1) = 1 - P\left(\bigcup_{n=1}^{z+1} \{N(w_n) = 0\}\right) \tag{3.2}$$

Computing $P(A)$ and $P(B)$ allows to compute $P(C)$, which is the probability of occurrence of a construction false positive ($P(C) = 1 - \frac{P(\bar{B})}{P(\bar{A})}$). Applied on a genome of size 10^{11}, we obtain the probability shown in Figure 3.1.

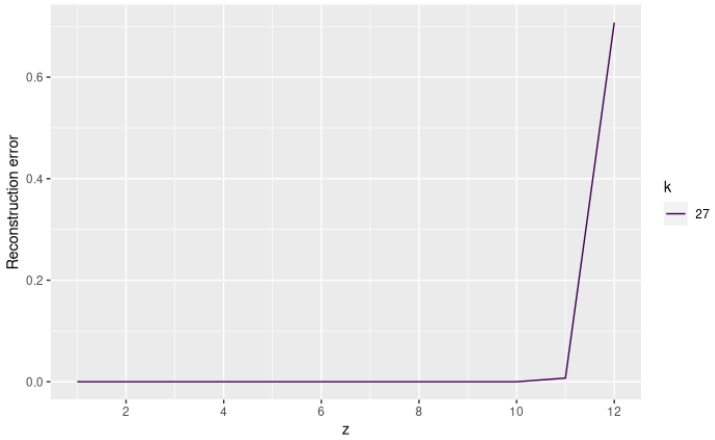


Figure 3.1: Probability of occurrence of a construction false positive with regard to z on a random genome of size 10^{11}, for multiple k values.

This indicates that using $s > 20$ is acceptable to index random genomes of up to 10^{11} bases since this generates a cFP rate of less than $10^{-7}\%$ for $k = 31$. In practice, genomes tend to have repeated regions (reducing their number of unique s-mers, which in turn limit the number of possible construction false positives). We thus recommend using $s > 20$ to index real datasets, as this generates a low false positive rate both in theory and in practice.

Note that on genomes of size 10^9, the sharp increase in the cFP rate (as shown in Fig 3.2) happens for a z value higher than $\frac{k}{2}$ (*i.e.* the cFP rate is very small for $z \leq \frac{k}{2}$).

As such, this statistical model cannot be applied to reproduce the sharp increase in the cFP rate for such genomes.

Since our publication about findere was accepted for publication before this analysis of its false positive rate, and because of the limitations of the limitation of $s > k/2$ (making it hard to use on a random genome smaller than 10^{11} for $k = 31$), we did not publish it.

3.5 findere's implementation

We propose an implementation of findere, available at https://github.com/lrobidou/findere, under the GNU Affero General Public License. Modifications, reuse and redistributions are allowed. Redistributions must include a copy of the modified source code. This implementation uses a Bloom filter as its inner AMQ. However, any other AMQ implementation can be used through a simple wrapper (provided with the findere implementation). The BF chosen for this implementation is a fork of the original https://github.com/mavam/libbf, which was modified to add the support of serialization. Although findere can index and query any alphabet, its implementation proposes a specialization for genomic sequences: as such, one can index not only natural language but also fasta and fastq files (gzipped or not) representing genomes or any sequencing read files. In this genomic context, a function to index and query canonical k-mers is also made available.

3.6 Results

We propose results on real biological data and natural texts. The aim is to show the practical advantages offered by findere, both in terms of query precision, index size, and query time. Being developed to be used on top of any AMQ, findere may be used for filtering the results from any such data structure, including Cuckoo Filters for instance. To the best of our knowledge, the only tool comparable to findere is kBF. We compared findere and kBF on biological data only as kBF is not designed to work on a generic alphabet.

Executions were performed on the GenOuest platform on a node with 4x8-cores Xeon E5-2660 2,20 GHz with 200 Go of memory. A complete description of tool versions, data acquisition, command lines, and numerical results are provided in the GitHub repository

`https://github.com/lrobidou/findere/tree/master/paper_companion`.

3.6.1 Experimental data

Metagenomic data. To measure the impacts of the `findere` algorithm on real genomic data, we used two HMP [3] fastq files, indexing reads from sample SRS014107 and querying reads from sample SRS016349, both downloaded from the NCBI Sequence Read Archive. These samples contain respectively 4.2 million reads of average size 92 characters and 2.3 million reads of average size 96 characters. We simply refer to this dataset as the "hmp" dataset.

Natural language data. To test the `findere` implementation on natural language, we used a dump of Wikipedia, from which we extracted two subsets overlapping with 10 Mo. They have each a size of 105 Mo, leading to about 10^8 31-mers each. We refer to this dataset as the "natural language dataset".

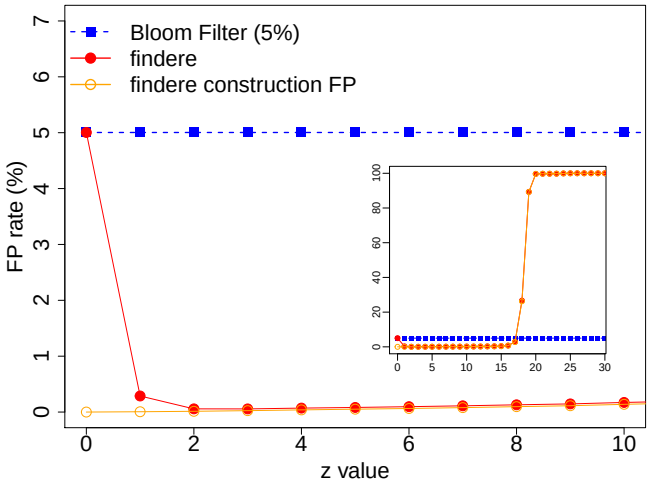
3.6.2 Results on genomics data

In this section, we analyze the effects of `findere` on the false positive rate and the space usage on an `AMQ` filter indexing genomics data. We provide results for $k = 31$ since it is a common value for k in bioinformatic tools.

False-positive analyses

We obtained results with a classical BF, indexing SRS014107 and querying SRS016349 with k-mers of length $k = 31$. For all experiences the size of the used BF is ≈ 2.6 billion bits, leading to 5% FPR when indexing 31-mers.

With the same BF size, `findere` was run with $k = 31$, varying the z value (and thus the s values). As shown in Figure 3.2, with $z = 0$, `findere` obtains, as expected, the same results as those obtained with the original `AMQ` filter. With low z values (*e.g.* lower than 5), the `findere` FPR quickly drops close to zero. For instance, with $z = 3$, $FPR_{\texttt{findere}}$ is equal to 0.056%, which is two orders of magnitudes smaller than the original FPR_{AMQ}. Also, with such low z values, s-mers are large enough to limit the `findere` "construction FP" (cFP) to negligible values. For instance, $z = 3$ leads to $s = 28$ and the cFP rate is 0.025%.



False positive rate, depending on z, $k = 31$

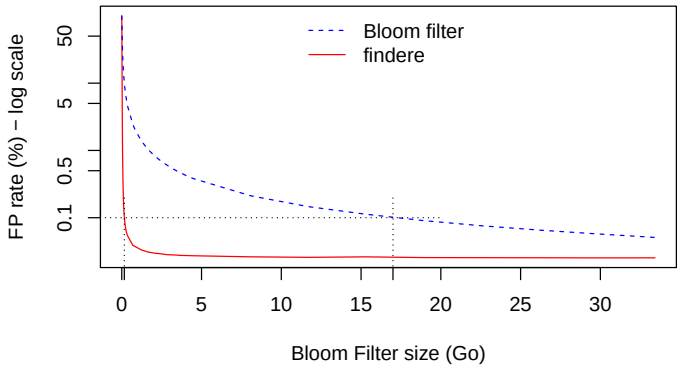
Figure 3.2: Comparative false-positive rate obtained on the hmp dataset. A BF indexes 31-mers, with a practical $FPR_{\texttt{AMQ}} =\approx 5\%$ (blue-filled squares, independent of the z value). With the same BF size, `findere` was run varying the z value (and thus the s value), leading to $FPR_{\texttt{findere}}$ as shown in red-filled circles. Orange empty circles show the amount of `findere` "construction FP", the rest of `findere` FP being due to a false positive in the underlying Bloom filter. The full figure zooms on recommended z values (in particular $z \in [2, 5]$). The small frame shows results including higher but discouraged z values.

When using large values of z (here > 15), s-mers get too small to be specific enough. Hence they have a high probability of randomly appearing in a given dataset, leading to a dramatic increase in the cFP rate. This happens with z values leading to s-mers of size < 13. For instance, $k = 31$ and $z = 19$ leads to $s = 12$, which results for `findere` in an FPR of 89.27%, being almost only composed of cFP (curves overlap on the figure, cFP representing 99.95% of the FP).

Fortunately, fixing z is easily anticipated by choosing a value small enough so that the indexed s-mers in the AMQ have a low chance (*e.g.* lower than 0.01%) to appear by chance. As stated in the theoretical analysis, the probability for a given s-mer to appear in a random text of length n can be roughly approximated as $1 - \left(1 - \frac{1}{|\Sigma|^s}\right)^n$, with $|\Sigma|$ being the size of the alphabet. For instance on the indexed SRS014107 sample, where $|\Sigma| = 4$ and $n = 386$ millions, the probability to appear by chance for a s-mer is respectively equal to 0.99% for $s = 13$ ($z = 18$), to 0.30% for $s = 15$ ($z = 16$), to 0.02% for $s = 17$ ($z = 14$), and to 0.005% for $s = 18$ ($z = 13$). Hence choosing $z < 14$ is acceptable. By default `findere` uses $z = 3$, as it is small enough to generate few cFP and high enough

to reduce the original $FPR_{\texttt{AMQ}}$ of two orders of magnitude.

Space gain



False positive rate, depending on the Bloom filter size, $k = 31$

Figure 3.3: `findere` and BF FPR depending on the space used, on the hmp dataset. The dotted line segment corresponds to 0.1% false-positive rate. Parameters used are $k = 31$ and the default value $z = 3$.

We recall first that `findere` memory usage has no overhead compared to the size of the used `AMQ` filter. For a given filter size, we can deduce the FPR for a BF and `findere`.

Results are presented Figure 3.3 (with the default $z = 3$ value). This can also be used for deducing the amount of space needed for a given FPR. For instance, to obtain a usual value of 1% FPR, `findere` requires 0.05 Gio of space while a BF requires 1.06 Go. The `findere` advantage gets even more important with a lower FPR: with 0.1% of FPR, `findere` requires 0.16 Go while a BF requires about 17 Go (dotted lines). This leads to a gain of space of two orders of magnitude, while not requiring any additional run time or RAM.

Query time

Thanks to the optimization detailed section 3.2.2, the query time decreases when z increases.

As shown Table 3.1, with discouraged values $z = 0$ or $z = 1$, the query time of `findere` is slightly higher than querying the original BF. This is due to additional conditional tests. With recommended z values ($z = 2$ to 5), compared to the query time of the BF, the

`findere` query time is divided by a factor 2 to 3. With the default $z = 3$ value, query time is divided by 2.4, while query time still decreases when z increases: with $z = 10$, the query time is divided by ≈ 5.

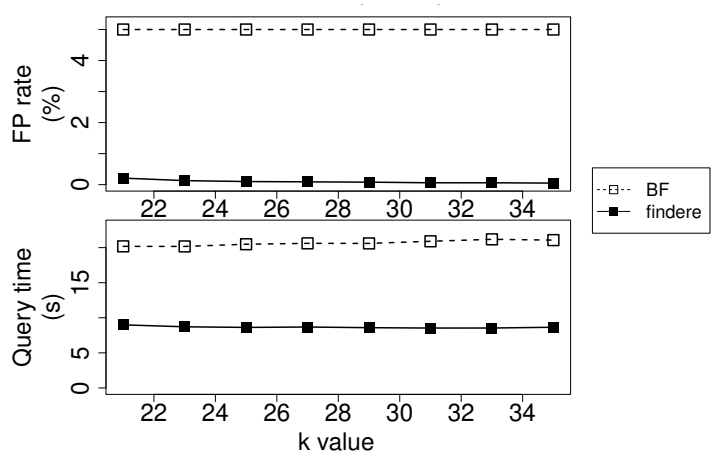
z	0	1	2	3	4	5	10
BF	42.4						
`findere`	42.9	43.7	24.3	17.5	14.1	12.0	8.6

Table 3.1: BF and `findere` query time in seconds on the hmp dataset (indexing sample SRS014107 and querying sample SRS016349), depending on the z value. Bloom filter's query time result does not depend on z and is reported only for $z = 0$.

Results varying k

In this section, we show that `findere` is robust with regard to the k value.

We varied the value of k and we reported the false positive rate as well as the query time. Tests were performed on the hmp dataset using the default $z = 3$ value. For each k value, we determined $size_k$ being the size of the Bloom filter such that its FPR for indexing k-mer is 5%. Then, for each k value, we applied `findere` on top of a Bloom filter of size $size_k$ and computed both the $FPR_{\texttt{findere}}$ and the query time. Results are presented Fig 3.4, along with results obtained with the original `AMQ` filter.



findere results depending on k

Figure 3.4: `findere` FPR and query time depending on k, on hmp dataset (indexing sample SRS014107 and querying sample SRS016349), using $z = 3$.

Results show that both `findere` and the original `AMQ` filter are robust with the k value, *i.e.* `findere` preserves this property of the underlying `AMQ` filter.

Comparison with `kBF` on the hmp dataset

`kBF` comes with two versions: 1-`kBF` and 2-`kBF`. 1-`kBF` uses less space at a higher FPR than 2-`kBF`, while 2-`kBF` uses an additional hash set, reducing the FPR at the cost of higher memory usage compared to 1-`kBF`.

The 2-`kBF` strategy imposes to query up to nine times the BF when asking for the membership of a single k-mer. At the same time, `findere` is ≈ 2.4 times faster than a Bloom filter with the recommended value $z = 3$. Hence, `kBF` is much slower than `findere`. For instance, for querying the hmp dataset, with a BF FP of 5%, 1-`kBF` (resp. 2-`kBF`) query needs $\approx$ 300s (resp. $\approx$ 1450s) while `findere` needs $\approx$ 17s.

To date, in the `kBF` implementation, k-mers are read from a file and indexed. A fraction of these indexed k-mers is mutated before being queried, to compute the false positive rate. The `kBF` implementation does not propose a way to index a set of reads and to query another set of reads. Furthermore, its implementation does not enable to serialize the index and does not enable setting or getting the size of the index.

Finally, it is not possible to pass directly the amount of space available for `kBF` to index a dataset. Instead, the user must provide an integer value (called "size factor"), which is the desired number of bits per element in the bloom filter used by `kBF`. Consequently, the space used by its bloom filter is always a multiple of the number of elements indexed. The size factor is decided at compilation time for 1-`kBF` and cannot be changed later on.

As a consequence, in order to be able to compare `findere` and `kBF`, we propose a fork of `kBF` (available here: `http://github.com/lrobidou/kbf/`) that allows to index a set of reads and query another one, as well as changing the size factor for 1-`kBF`. Then, using the size factor, we were able to determine space and memory 1-`kBF` would end up using. However, 2-`kBF` needs to store an additional hash set of k-mers. Since we were not able to get the amount of space taken by that hash set, we do not report it here. Consequently, the "size required" column in Table 3.2 is to a great extent underestimated for 2-`kBF`. Indeed, the peak RAM usage is between 7 Go and more than 8 Go for all experiments proposed in this Table.

From Table 3.2, we can observe that `findere` outperforms Bloom filters, 1-`kBF` and 2-`kBF` in terms of false positive rate. For a high number of bits per element, the FPR of `findere` and 2-`kBF` seem close, however, comparisons with 2-`kBF` are not fair as it also

number of bits per element[1]	size required (Go)[1]	FPR BF (%)	FPR 1-kBF (%)	FPR 2-kBF (%)	FPR findere (%)
3	0.05	28.34	26.38	15.40	0.95
5	0.09	18.12	14.49	5.52	0.27
7	0.12	13.32	9.10	2.54	0.15
9	0.15	10.52	6.22	1.37	0.11
15	0.26	6.45	2.69	0.36	0.07
18	0.31	5.41	1.96	0.22	0.06
21	0.38	4.65	1.49	0.14	0.05
24	0.41	4.08	1.17	0.10	0.05
26	0.44	3.77	1.01	0.08	0.05

[1]**Not** including the hash set for 2-kBF

Table 3.2: FPR of Bloom filter, 1-kBF, 2-kBF and findere with respect to the size of the filter, for k=31, on the hmp dataset (indexing sample SRS014107 and querying sample SRS016349). findere was used with default $z = 3$ value.

computes a hash set of k-mers, which space is not included in the Table. This set leads to an unreasonable space usage (*e.g.* 7.78 Go for an FPR of ≈1% when findere requires 0.05 Go).

As it shows a higher FPR for a fixed amount of space, kBF uses more space than findere for an equivalent FPR. For instance, with ≈ 1% FPR, findere requires 0.05 Go of space while 1-kBF requires ≈ 0.40 Go.

In Table 3.3, we show that, for the indexation and the query steps, both 1-kBF and 2-kBF are slower than the Bloom filter used in the implementation of kBF.

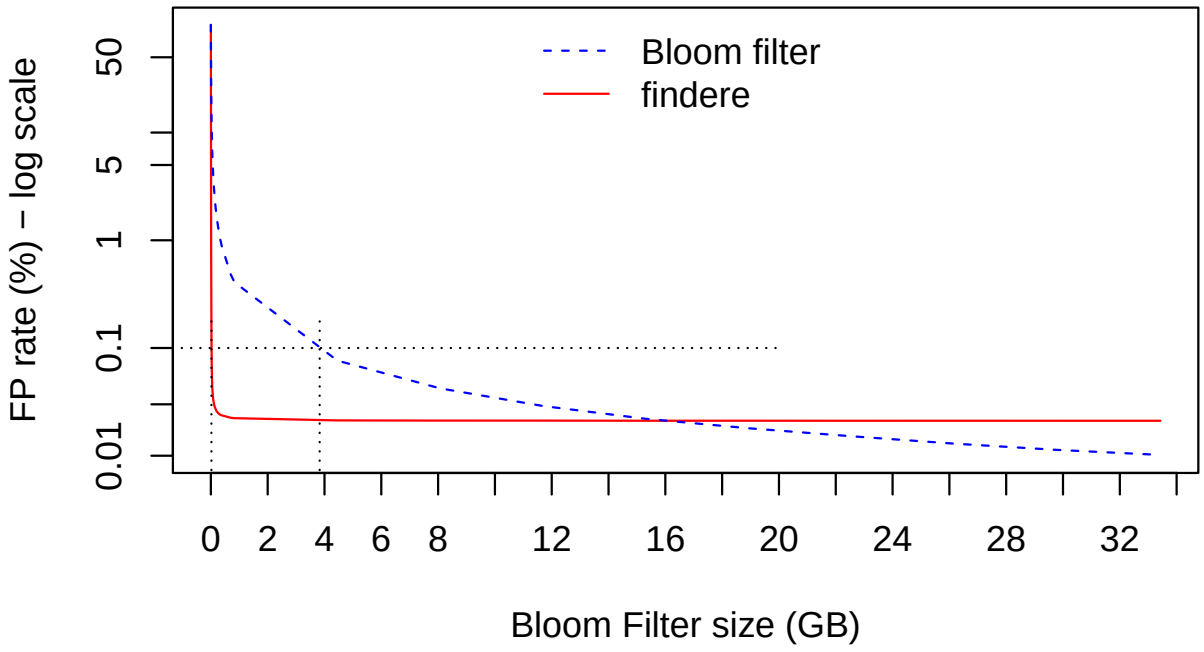
size factor[1]	3	9	21
time index bf (s)	223	242	250
time query bf (s)	178	207	217
time index 1-kBF (s)	220	241	248
time query 1-kBF (s)	329	332	304
time index 2-kBF (s)	506	419	355
time query 2-kBF (s)	1399	1440	1453

[1]The size factor only takes into account the bloom filter used by kBF, not the adjacent hashSet

Table 3.3: Time taken (in second) by kBF to index and query all of the 31-mers in the hmp dataset.

3.6.3 Results on natural languages

In this section, we show results obtained when applying `findere` on the natural language dataset. As stated in Section 3.6.1, in this setup, both the indexed and the queried data consist of about 10^8 31-mers extracted from Wikipedia. Figure 3.5 shows the FPR of `findere` and Bloom filters when indexing 31-mers from this dataset depending on the size of the `AMQ` filter. Table 3.4 shows the time taken by a Bloom filter and `findere` to query this dataset with respect to the value of z.



False positive rate, wikipedia corpus, depending on the Bloom filter size, $k = 31$, $z = 3$

Figure 3.5: `findere` and BF FPR depending on the space used by their index, on the natural language dataset. The dotted line segment corresponds to 0.1% false positive rate.

z	0	1	2	3	4	5	10
BF	19.36						
`findere`	19.87	19.99	11.76	8.77	7.32	6.56	5.56

Table 3.4: BF and `findere` query time in seconds on the natural language dataset, depending on the z value. BF result does not depend on z and is reported only for $z = 0$.

Memory gain. As in Section 3.6.2, we computed the FPR with regard to the space used. From those results, we can deduce that on natural languages, for an FPR of 0.1%, the `findere` space usage is two orders of magnitude less than BF. Indeed, `findere` needs 0.023 Go, while a BF requires 3.38 Go. These results are consistent with the results on the hmp dataset. Interestingly, Figure 3.5 shows a limit of `findere`: the construction false positive rate is a lower bound of $FPR_{\texttt{findere}}$. As such, when using more than a certain amount of space (in this setup: 15 GB), the `AMQ` filter will yield a false positive rate lower than `findere`. Recall that the set of words used in natural languages is orders of magnitude smaller than the set of words in genomic datasets, which may explain why the theoretical limit of `findere` is reached for an `AMQ` filter size of 15 GB in this setup (while it is not in the metagenomic setup).

Query time. As described in section 3.2.2, the query time decreases when z increases. This holds when `findere` is used on the natural language dataset as well. With recommended z values ($z = 2$ to 5), compared to the query time of the Bloom filter, the `findere` query time is divided by a factor 1.6 to 3. With the default $z = 3$ value, query time is divided by 2.2 compared to the raw Bloom filter query. Results are consistent with the metagenomic setup: `findere` speeds up queries with regard to z. With the recommended parameter $z = 3$, `findere` allows performing queries two times faster compared to the original `AMQ` filter.

3.7 Implementation

We propose an implementation of `findere`, available at `https://github.com/lrobidou/findere`. It exposes a simple and easy-to-use command line interface. Parameters include:

- a list of files to index
- the name of the output index
- the size of the `AMQ` filter
- the value of k
- the value of z
- the type of data to index and query (whether biological data or not)

- whether to use canonical words
- a threshold, to only display queries having a proportion of present k-mers above the threshold

Usage is straightforward:

```
./bin/findere_index -i <lists of files to index>
                    -o <index_name>
                    -b <size in bits>
                  [ -K <k> ]
                  [ -z <z> ]
                  [ -t <type of data {bio/text}> ]
                  [ -c <whether to use canonical words or not>]

./bin/findere_query -i <index_name>
                    -q <your query file>
                  [ -t <{bio/text}> ]
                  [ -c <whether to use canonical words or not>]
                  [ --threshold <threshold> ]
```

This implementation also proposes a simple wrapper to use any other `AMQ` filter implementation. Technically, any `AMQ` filter class inheriting from the virtual class `customAMQ` and exposing a method `contains` is compatible with our implementation.

3.8 Conclusion

In this Chapter, we proposed a strategy for reducing the query time and false positive rate of any approximate member query (`AMQ`) data structure when used for querying words of length k from a query. This strategy can be applied no matter the alphabet. Despite its amazing simplicity, applied on metagenomics and natural text data, compared to the original `AMQ` (a Bloom filter with one hash function): `findere` **1/** makes queries two times faster, **2/** enables to decrease by two orders of magnitude the false-positive rate or enables to decrease the space allocated to each element by two orders of magnitudes, and **3/** has no impactful drawback when used with recommended values.

The `findere` algorithm has been implemented in kmtricks [4], reducing its false positive rate when indexing *Tara* oceans metagenomic datasets.

In the next section, I will describe another of our contributions, `fimpera`, which allows retrieving counts' information of indexed elements.

Chapter 4

Second main contribution: `fimpera`

This chapter corresponds to our published paper in Bioinformatics, 2023 [28]. Sections of the original paper were rewritten to add clarity and to match the structure of the manuscript.

4.1 Context

In the previous chapter, I presented one of our contributions, `findere`, which is a simple strategy for reducing the false-positive rate of any AMQ data structure indexing reads through the presence-absence of their constituent k-mers. We first focused on improving methods indexing presence-absence of k-mers because these methods play a crucial role in bioinformatics.

However, apart from presence-absence information, the information about the abundance is also of high interest. This information is crucial for many biological applications such as detecting cancer-related mutations [64] transcriptomics, metagenomics, or metatranscriptomics analyses. As such, the focus of our second contribution is to record the abundance of indexed k-mers, not just their presence-absence.

At first sight, methods like the counting Bloom filter [26] seem to be one way to index the abundance of indexed elements. However, its non-null false positive rate and its tendency to overestimate the abundance of k-mers are major downsides.

The counting Bloom filter is one of the data structures indexing counts of indexed elements with a non-null false positive rate and overestimation rate. In this manuscript, we call those data structures "couting Approximate Membership Query filters" (`cAMQ` filters for short).

Instead of developing a novel `cAMQ` filter, we rather focused on improving existing ones. In this chapter, we propose a wrapper to improve any existing `cAMQ` filter. The method

we introduce is called fimpera. It generalizes our previous contribution, findere.

The fimpera algorithm can be used on top of any query made using any kind of cAMQ filter. However, our implementation and tests are proposed on top of a counting Bloom filter (cBF) that uses a unique hash function. This choice is motivated by the fact that cBFs are the simplest and are among the most widespread data structures for dealing with billions of elements. The number of hash functions is chosen to stay consistent with state-of-the-art methods indexing presence-absence information in AMQ filters (like HowDeSBT [61] for instance).

fimpera is publicly available at https://github.com/lrobidou/fimpera under the GNU Affero General Public License. Modifications and redistributions are allowed as long as distributions of modified implementation include a copy of the modified code.

Additionally, the algorithmic needs of fimpera led us to propose a novel algorithm for computing the sliding window minima (resp. maxima): the minimal (resp. maximums) values of all sub-arrays of a fixed size over an array v of $|v|$ values. This algorithm runs in $\mathcal{O}(|v|)$ time and requires no dynamic memory allocation. This contribution may be useful outside the fimpera context. Its novelty is that being in place, it uses no additional memory, unlike other approaches. This makes it the fastest known algorithm to perform this task.

This contribution is available at https://github.com/lrobidou/sliding-minimum-windows along with a benchmark comparing it to other solutions. Its license is the same as fimpera.

In the next sections, we will describe fimpera: its core principle, its indexation and query steps, and the way we measure its effect on the false positive rate and overestimation rate of the wrapped cAMQ filter. Then, we will present an algorithm for computing the sliding window minima of a vector. Finally, we will present the results of fimpera and the sliding window algorithm.

4.2 Notations and metrics

In this chapter, we consider a dataset composed of a multiset of sequences. Given a dataset D, $\mathcal{D}_k$ denotes the multiset of k-mers extracted from D.

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

A cAMQ filter represents a multiset of elements $\mathcal{D}_k$. It can be queried with any element

d; the query's response on a `cAMQ` filter, denoted by $query(d)$, is always either correct or overestimated, i.e. $query(d) \geq abundance(\mathcal{D}_k, d)$. If $query(d) = abundance(\mathcal{D}_k, d)$, the `cAMQ` filter reports the correct abundance, otherwise it reports an overestimation. Note that underestimation is not possible.

To measure the quality of the `fimpera` results and the underlying `cAMQ` filter results, we propose three metrics: the false positive rate (FPR_{cAMQ}), the overestimation rate ($overest_{cAMQ}$) and the overestimation score. These metrics are defined in the following sections.

4.2.1 False positive rate

If $abundance(\mathcal{D}_k, d) = 0$ and $query(d) > 0$, then d is found in the `cAMQ` filter even if it is absent from $\mathcal{D}$. This case is a false positive call. The false positive rate of a `cAMQ` filter, 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 ($n = 0$).

4.2.2 Overestimation rate

If $query(d) > abundance(\mathcal{D}_k, d) > 0$, then the `cAMQ` filter reported an overestimation. We define the overestimation rate as $overest_{cAMQ} = \frac{\#OVER}{\#OVER+\#CORRECT}$, $\#OVER$ and $\#CORRECT$ denoting respectively the number of correctly reported abundance and the number of overestimated calls among a query.

$overest_{cAMQ}$ is the ratio of overestimated calls among true positive calls. $overest_{cAMQ}$ is only defined on true positive calls ($abundance(\mathcal{D}_k, d) > 0$) to better evaluate the effect of our method. This way, improving $overest_{cAMQ}$ measurement won't interfere with the measure of FPR_{cAMQ} since they are not evaluated on the same `cAMQ` filter calls.

4.2.3 Overestimation score

When evaluating a method, we set up a metric that we call "overestimation score". It is defined as the sum of the square of errors made on the output of each method:

$$OS = \sum_{i=0}^{i<|Q|-k+1} (query(Q[i:i+k]) - abundance(\mathcal{D}_k, Q[i:i+k]))^2$$

The lower the overestimation score is, the fewer errors were made and/or the closer the errors are to the ground truth. Errors that are closer to the ground truth are less penalized by this score than errors distant from it.

Both FPR_{cAMQ}, $overest_{cAMQ}$ and the overestimation score depend on the used `cAMQ` filter strategy and on the amount of space used by this `cAMQ` filter.

4.3 Overview of `fimpera`

4.3.1 Core principle

`fimpera`'s objectives are to reduce FPR_{cAMQ}, $overest_{cAMQ}$ and the overestimation score of a `cAMQ` filter.

Just like `findere`'s strategy, this is achieved using a method based on splitting k-mers into smaller words called s-mers.

Indeed, the `fimpera` strategy comes from the following observations:

- If a k-mer is present in a dataset x times, then all its subwords are also present in the dataset at least x times.
- Conversely, if one of its subwords is present at most y times in a dataset, a k-mer is present at most y times in that dataset.

As for `findere`, in the following, we set $z = k - s$, hence $z \geq 0$.

The s-mers are the elements indexed in the `cAMQ` filter at indexation time. At query time, a k-mer is reported as present x times if and only if the least abundant of its s-mers is present x times in the `cAMQ` filter. For a k-mer d:

$$query(d) = min(query\_smer(smer), \forall smer \in d)$$

A false positive (resp. overestimation) on a k-mer requires up to $z+1$ false positives (resp. overestimations) on its s-mers.

- *Proof: Consider a k-mer present x times in a dataset having all its s-mers present*

x times in this dataset

- ◊ *If p (p < z + 1) s-mers are overestimated*: There are still $z + 1 - p > 0$ s-mers having an abundance of x. As such, the minimum of abundance of s-mers is x: the k-mer is not overestimated.

- ◊ *If p (p = z + 1) s-mers are overestimated*: There are $z + 1 - p = 0$ s-mers having an abundance of x. As such, the minimum of abundance of s-mers is strictly greater than x: the k-mer is overestimated.

- Thus, overestimating $z + 1$ s-mers is required to overestimate such k-mer.

As such, `fimpera` requires more stringent conditions to return a false positive k-mer (or an overestimated k-mer) compared to the underlying `cAMQ` filter. Alternatively, for a fixed FPR_{cAMQ}, the `fimpera` strategy reduces the memory needed by the couting Approximate Membership Query filter up to several orders of magnitudes.

4.3.2 Indexation overview

At indexation time, `fimpera` takes as input a file of counted fixed-length words, as provided for instance by KMC [65]. `fimpera` can either take counted k-mers or counted s-mers as input. Taking counted s-mers is straightforward: each indexed s-mer is indexed in the `cAMQ` filter with its abundance. If the input consists of counted k-mers, `fimpera` splits each k-mer into its $k - s + 1$ constituent s-mers ($k \geq s$). Each s-mer is then indexed in a couting Approximate Membership Query filter along with its so-called s-abundance, denoted as s_{ab}. The s_{ab} of a s-mer is formally defined as the maximum of the abundance of the k-mers containing this s-mer. We explain the choice of relying on s_{ab} instead of the abundance of s-mers and describe its outcomes in Section 4.7.11. Using both the s_{ab} and the abundance of s-mers is supported in the implementation.

4.3.3 Query overview

The query algorithm described here mentions the concept of s_{ab}. Note that it is fully compatible with the use of s-mers' abundance (which is not mentioned here for the sake of brevity).

For each queried sequence Q, `fimpera` extracts its sequence of s-mers, which are then queried against the couting Approximate Membership Query filter, and the abundance

of any k-mer of Q is computed as the minimum of its s-mers' s_{ab}. By default, `fimpera` outputs each input sequence along with the abundance of every of its consecutive k-mer. In practice, the input query file is a fasta or a fastq file, possibly gzipped. Users may also change the output of `fimpera` to match their biological needs, *e.g.* output the average of the abundance of k-mers for each input sequence.

4.3.4 False positive calls

Let's consider a k-mer d with an abundance of 0 and each of its s-mer has an s_{ab} 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 `cAMQ` filter. The probability of such an event is ${FPR_{cAMQ}}^{z+1}$, leading in this case 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 the default value) and a `cAMQ` filter having a false positive rate of 25%, the false positive rate with `fimpera` for that setting is $\approx 0.4\%$.

As `findere`, the `fimpera` approach may generate a novel kind of false positive, which we call "construction false positive". These false positives are created *independently of the underlying couting Approximate Membership Query filter*. This event is non-null but it is in practice negligible when using usual k and z values, as shown in the results.

4.3.5 Overestimations

To overestimate the abundance of a queried k-mer with `fimpera`, overestimations are required to happen on the abundance of the less abundant s-mers of a k-mer. In practice, multiple s-mers of a k-mer may have the lowest abundance, so overestimations are required to happen on **every** one of them. The more s-mer per k-mer, the more s-mer abundance overestimations need to happen to overestimate a k-mer abundance. Using s_{ab}, s-mer abundance overestimations can 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 from the least abundant k-mer. This happens no 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 d_0 and d_1 overlapping over $k-1$ characters. If d_1 has an abundance greater than d_0, then the correct abundance of d_0 is retrievable through a unique s-mer (the unique s-mer of d_0 that does not appear in the k-mer d_1). In such case, d_0 is more likely to be overestimated than d_1.

Consequently, `fimpera`'s overestimations are not uniformly distributed random events. Overestimations are more likely to occur close to a change in abundance along queried sequences than in a random k-mer. In such cases, those overestimations are limited to the abundance of their neighbor k-mers, mitigating their impact. Indeed results Section 4.7 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 detail both the indexing step and the querying step of `fimpera`, as well as one optimization, allowing a query time independent from the z value.

Another optimization enables `fimpera` to perform queries slightly faster than using the original underlying couting Approximate Membership Query filter by skipping querying some s-mers. This "skip" optimization is borrowed from `findere` and is described Section 3.2.2.

4.4 Querying with `fimpera`

`fimpera`'s query consists in querying all consecutive, overlapping k-mers from a sequence Q of size greater than or equal to 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 `cAMQ` filter and are stored in an array of integers s_{abs};
- the abundance of any k-mer starting position $p \in [0; |Q| - k - 1]$ is the minimum value of the sub-array of length $(z+1)$ starting at the position p: $s_{abs}[p; p+z]$.

A non-optimized version of the `fimpera`'s query algorithm is shown in Algorithm 4. This algorithm takes a queried sequence Q, a `cAMQ` filter indexing s-mers, and parameters k and z. It returns a vector of integers called $response$, such that: $\forall i \in [0, |response| - 1]$,

$response[i]$ is the abundance of the k-mer starting at position i in the query, for all i in $[0, |Q| - k + 1]$.

Algorithm 4 Non optimized `fimpera`'s query

1: **procedure** NON_OPT_QUERY($Q \in \Sigma^*$; `cAMQ` filter indexing s-mers; k and z in $\mathbb{N}^+$ with $|Q| \geq k, z < k$.)
2: ▷ Store the s_{ab} of all s-mers from Q:
3: $s_{abs} \leftarrow emptyVector(|q| - s + 1)$
4: **for** i in $[0; |Q| - s + 1]$ **do**
5: $s_{abs}[i] \leftarrow get\_s_{ab}(q[i, i + s - 1])$
6: ▷ get_s_{ab} uses the cAMQ to provide the s_{ab} of a s-mer
7: **end for**
8: ▷ Compute all k-mers abundances from s-mers s_{ab}:
9: $response \leftarrow emptyVector(|Q| - k + 1)$
10: **for** i in $[0; |Q| - k + 1]$ **do**
11: $response[i] \leftarrow \min_{j \in [i, i+z]}(s_{abs}[j])$
12: **end for**
13: **return** $response$
14: **end procedure**

Algorithm 4 is not optimal. Line 11 computes the minimal value of a range of z consecutive integers taken from a vector of integers. At each iteration of the *for loop* line 10, this range is shifted by one.

This step is improved by avoiding recomputing the minimal value of windows of length $z + 1$ starting at each position of an array. This optimization is described in the following section.

4.5 Optimisation: sliding window minimum algorithm

The problem, independent of `fimpera`, is as follows: given a vector of values (integers or floats) v and an integer W (denoting the size of a sliding window, with $W \leq |v|$), give an array r such that $\forall i \in [0, |v| - W], r[i] = min(v[i], v[i + 1], ..., v[i + W - 1])$. This is a particular case of a more general problem, the "range minimum query" (RMQ). Given a vector v of element drawn from a totally ordered set and two integers i, j ($0 \leq i < j < |v|$), the RMQ consists in finding $min(v[i], v[i+1], ..., v[j-1])$. Answering RMQ generally relies on some pre-computation beforehand (e.g. pre-computing the whole set of possible queries, but less resource-intensive solutions can be found, such as [66]).

The sliding window minimum problem studied here is a particular case of an RMQ ($j-i$ is fixed and queries consist in every window of that size). Thus, its solutions do not require taking into account other window sizes, effectively allowing to skip pre-computation. The naive approach to solving the sliding minimum problem is to simply search for the minimal value in each window. This algorithm is in $\mathcal{O}(W \times |v|)$ time. Some straightforward faster solutions can be built on top of dynamic heaps. Here, we propose a solution (named "fixed window") that does not rely on any heap allocation (which in practice is slow for most systems). This solution, without any dynamic memory allocation, is an order of magnitude faster than other $\mathcal{O}(|v|)$ solutions, as shown Section 4.8.

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

- min_L_j: $min_L\_j[i\%W]$ contains the minimum value encountered in the j-th fixed window up to the position i;$(i \in [i \times W \times j; i \times W \times (j+1) - 1]$
- min_R_j: $min_R\_j[i\%W]$ contains the minimum value from the position i up to the end of the j-th fixed window.

All min_L_j and min_R_j vectors are then concatenated into two vectors (min_L and min_R). The minimum of a *sliding* window starting at position i, denoted by $r[i]$, is thereupon the minimum between:

- $min_L[i + W - 1]$ (the minimum of the left part of the next fixed window)
- $min_R[i]$ (the minimum of the right part of the current fixed window)

An example is provided in Table 4.1.

Note that 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. 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 $min_L[i]$ on the fly (if $i\%W \neq 0$ then $min_L[i] = min(min_L[i-1], v[i])$, else $min_L[i] = v[i]$). **2/** min_R is computed *directly in the queried vector*. This does not impact the correctness of the algorithm, as $r[i] \leq min_R[i] \leq v[i]$. **3/** the response (minimal value per sliding window) can be stored directly in the input

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_L	5	3	3	$\underline{1}$	1	1	3	2	2	3
min_R	3	$\underline{3}$	7	1	4	5	2	2	2	3
r	3	$\underline{\mathbf{1}}$	1	1	3	2	2	2		

Table 4.1: Computation example of the r vector, with a window of size 3. Tables min_L and min_R are represented for helping comprehension but are not implicitly created in practice. The j row indicates the starting positions of each fixed window. As an example, the minimal value of the sliding window of size 3 starting position $i = 1$ is $r[1] = 1$ (bold underlined value), being equal to $\min(min_L[1 + 3 - 1], min_R[1]) = \min(1, 3)$ (underlined values).

queried vector as well. At the price of modifying the input vector, this allows the algorithm to be run in $\mathcal{O}(|Q|)$ time while avoiding any heap allocation, which is time-consuming.

Algorithm 5 details our proposal for computing all minimal values of a sliding window of a vector of integers v in linear time and with zero memory allocation. Its usefulness is not limited to `fimpera`. As so, we propose an independent implementation, available at `https://github.com/lrobidou/sliding-minimum-windows`. Note that modifying this algorithm for computing the maximal values instead of minimal values is straightforward. In Section 4.8, we propose results showing the advantages of this algorithm.

Algorithm 5 sliding_minimum_window

procedure SLIDING_MINIMUM_WINDOW(vector v of positive integers; length of window w ($|v| \geq w, w > 1$))
  $nbWin \leftarrow \lfloor size(v)/w \rfloor$
  $nb\_elem\_last\_window \leftarrow |v| \mod w$
    ▷ Computation of min_left (See Section 4.5)
  $min\_left \leftarrow v[0]$
  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
    $start\_window \leftarrow i \times w$
    for $index$ in $[start\_window + w - 2; start\_window]$ **do** ▷ decreasing order
      $v[index] \leftarrow min(v[index + 1], v[index])$ ▷ compute min_right, directly in v
    end for
    for j in [0; w-1] **do**

```
15:              ▷ sliding_minimum is min(min_left, min_right(=v)):
16:            v[start_window + j] ← min(v[start_window + j], min_left)
17:              ▷ update min_left (if a new fixed window starts, reset min_left):
18:            if j == 0 then
19:              min_left ← v[start_window + w]
20:            else
21:              min_left ← min(min_left, v[start_window + w + j])
22:            end if
23:          end for
24:        end for
25:          ▷ Computation for the last window is not described here for the sake of simplicity
26:          ▷ remove last w − 1 elements from v:
27:        for i in [0; w − 2] do
28:          v.pop_back()
29:        end for
30:        return v
31:    end procedure
```

4.6 Implementation of `fimpera`

An implementation of `fimpera` is available at `https://github.com/lrobidou/fimpera`. This implementation is specialized for genomic data (i.e. with an alphabet consisting of A, T, C, G) and uses a counting Bloom filter with a unique hash function as couting Approximate Membership Query. A template mechanism allows the use of any other couting Approximate Membership Query provided by the user. Queries consist of fasta or fastq files (gzipped or not), and an option is provided to index and query canonical k-mers only. Indexing options include the k and z values, the size of the filter, and b, the number of bits per element used to store its abundance. 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 $2^b - 1$ (to prevent overflow, the abundance is capped at $2^b - 1$).

In order to encode large abundance values with few bits per k-mer, instead of storing the first possible $2^b - 1$ distinct abundance values, we propose to discretize any abundance value to user-defined interval ranges. In practice, `fimpera` can use any surjective function

$$f(x \in [1, \infty]) \rightarrow y \in [1, 2^b - 1],$$

supplied by the user. The proposed implementation proposes the usage of $y = \min(\lfloor \log_2(x) + 1 \rfloor, 2^b - 1)$, or $y = \min(\lfloor \log_{10}(x) + 1 \rfloor, 2^b - 1)$. Note that $x = 0$ corresponds to the abundance of non-indexed s-mers, and thus is not taken care of by our implementation.

Changing the default output (e.g. storing results instead of printing them, computing average abundance per sequence, or printing only sequences whose average k-mer abundances is above a user-defined threshold) is possible.

4.7 Results of `fimpera`

4.7.1 Experimental setup

To the best of our knowledge, no other tool focuses on reducing the false positive rate of existing couting Approximate Membership Query filters, thus we compare `fimpera` results applied on a counting Bloom filter indexing s-mers with the original counting Bloom filter results indexing k-mers using a unique hash function. We propose results on biological marine metagenomic data, described below.

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 GB of memory.

4.7.2 Metagenomic dataset

We used two fastq files from the *Tara* ocean metagenomic dataset [5] to show the advantages offered by `fimpera` on metagenomic samples. The index was computed from the 2.38×10^8 distinct 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.

4.7.3 Choice of filters parameters

In this experiment, 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 element for storing

the abundance of indexed k-mers. For indexing 2.38×10^8 31-mers, this structure requires 6.96×10^8 slots (thus 3.48×10^9 bits).

Abundances are stored as their $\lfloor \log_2 \rfloor$ values. We use the default $z = 3$ parameter (unless otherwise stated). As we use $z = 3$, we compare the 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).

4.7.4 False positive rate analyses

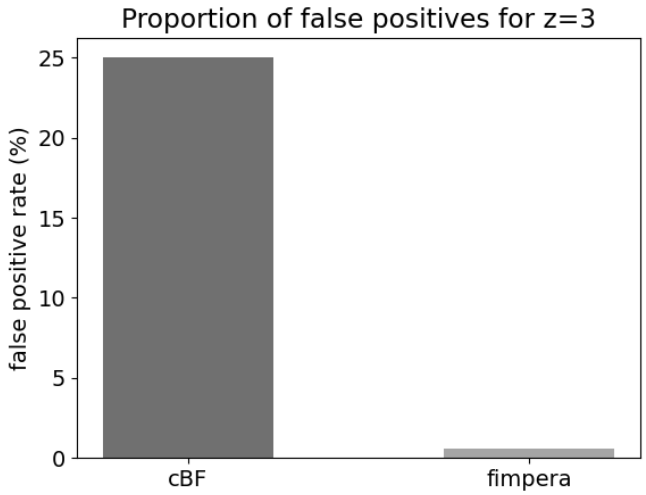


Figure 4.1: Proportion of false positive calls without `fimpera` (on a classical counting Bloom filter) and with `fimpera` ($z = 3$), indexing and querying real metagenomic read datasets.

Results about false positives obtained with the proposed experiment are shown in Fig. 4.1. Results about the `cBF` simply confirm the setup and show 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 4.3.3), 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 Table 4.2).

4.7.5 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`.

Results comparing the proportion of calls reported with an incorrect abundance among the true positives 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 "construction overestimation".

4.7.6 Distribution of errors in overestimated calls

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

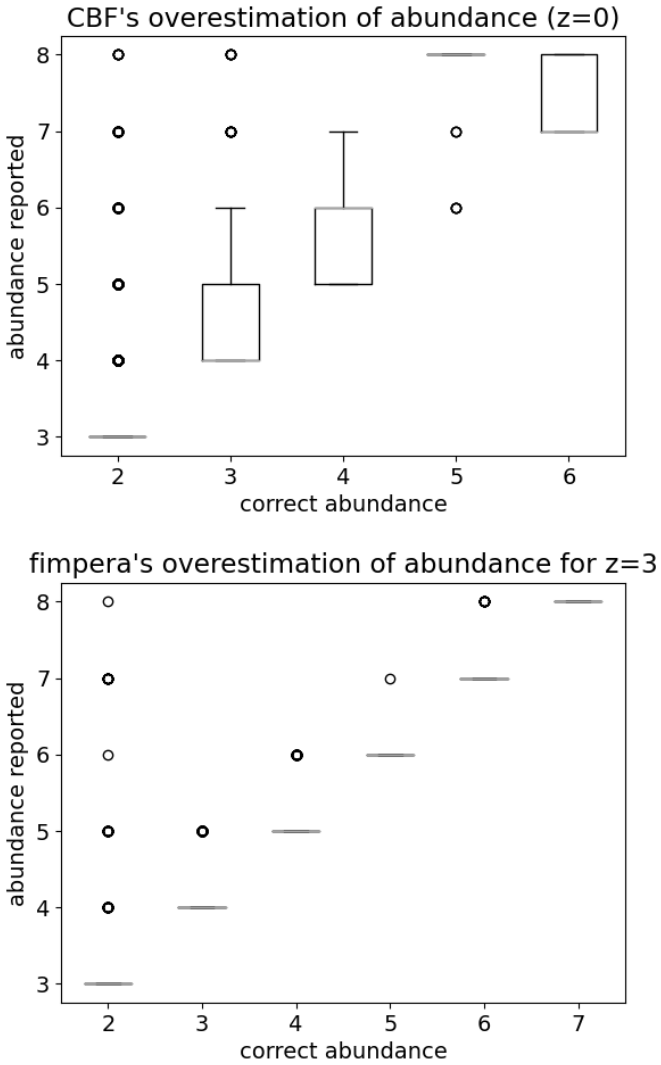


Figure 4.2: For true-positive calls with an incorrect abundance estimation: reported abundance with respect to the correct abundance. Top: using the original `cBF`, Bottom: using `fimpera`.

Results presented Fig. 4.2 show that, as stated in Section 4.3.5, the erroneous abundance calls are closer to the ground truth with `fimpera` compared to those obtained with the original `cBF`. As seen Fig. 4.2-bottom, with `fimpera`, almost all (except for a few outliers) overestimations are only one value apart from the correct range (the average difference with the correct abundance range is 1.07). With the original `cBF`, as seen Fig. 4.2-left, overestimations are more important (1.33 range in average from the ground truth).

4.7.7 Influence of the size of the underlying couting Approximate Membership Query filter.

Because there is a trade-off between space usage and the false positive rate of a couting Approximate Membership Query filter, `fimpera` can either reduce the false positive rate of the couting Approximate Membership Query filter without changing its size or reduce its size without changing its false positive rate (or set a trade-off between these two metrics). This section focuses on the reduction of the size of the underlying couting Approximate Membership Query filter for a given false positive rate.

This trade-off (with and without `fimpera`) is shown Fig. 4.3 and Fig. 4.4. To be as close as possible to real-life use-case, we consider in this section the average abundance of k-mer on each read, not the abundance of each k-mer. In order to precisely assess the overestimations, abundances were not stored as their $\lfloor \log_2 \rfloor$ values, but rather as their original values. Values were thus stored on 8 bits, to cap the maximum value at $2^8 - 1$, instead of $2^5 - 1$.

Here, we evaluate the false positive rate and the overestimation score yield by both the counting Bloom filter and `fimpera`.

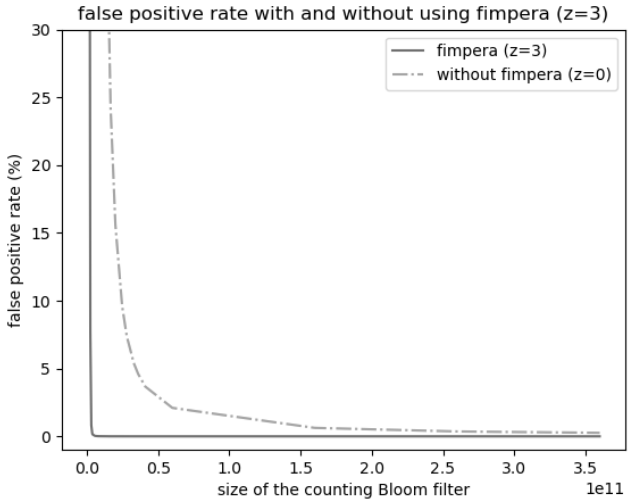


Figure 4.3: Variation of the false positive rate with respect to the size (in bits) of the `cBF`, with and without `fimpera` ($z = 3$). As we consider here the average abundance of k-mers on each queried read, a false positive means that the average abundance of a k-mer on a read is strictly positive while it should have been 0.

With fimpera, achieving a false positive rate of 2% requires $\approx 2.98 \times 10^9$ bits ($\approx 3.72 \times 10^8$ slots of 8 bits each) (see Fig. 4.3). Without fimpera, achieving the same false positive rate requires $\approx 65.5 \times 10^9$ bits (21 times more space). The lower the false positive rate, the greater the space gap between using and not using fimpera. Achieving a false positive rate of 0.02% requires about 7.25×10^9 bits with fimpera, but 360×10^9 bits were not enough without fimpera (i.e. allocating more than 50 times the space budget of fimpera would be required).

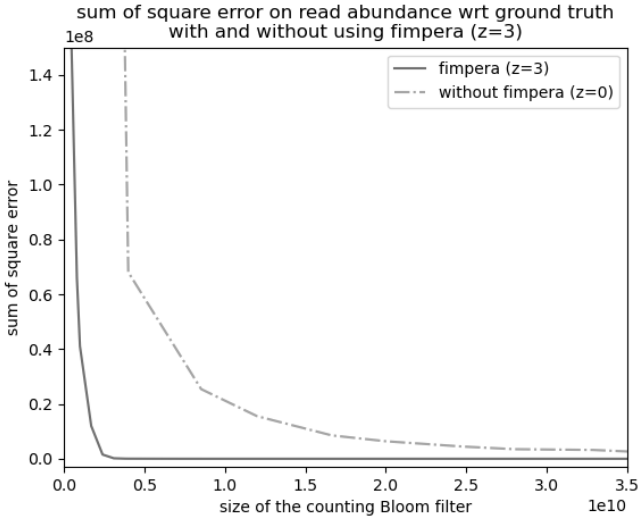


Figure 4.4: Sum of square error with respect to the size (in bits) of the cBF, with and without fimpera ($z = 3$). In this experiment, the output of fimpera is the *average of each input read*.

Fig. 4.4 shows the sum of square of overestimations, with and without fimpera. For an overestimation score (defined Section 4.2.3) of 200000, fimpera requires $\approx 3.10 \times 10^9$ bits ($\approx 3.87 \times 10^8$ slots of 8 bits each), whereas a cBF would requires 330.9×10^9 bits (i.e. 100 times more space).

4.7.8 Impact of z

In this section, we show the impact of the z value on the quality of the results of fimpera.

As shown in Table 4.2, **for a given k value**, the false positive rate decreases with respect to z and stays low for a wide range of z values (at least from $z = 3$ to $z = 9$ when using $k = 31$). When using an extreme z value, for instance, $z = 20$, the false positive rate

z	0	3	5	7	9	20
(indexed s-mer size $s = 31 - z$)	31	28	26	24	22	11
False positive rate (%)	25.00	0.56	0.08	0.02	0.03	99.70
Of 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
Of which: constr. overest. (%)	0	83.02	88.65	94.24	94.34	99.85
Query time (s)	557	514	498	491	493	528

Table 4.2: Influence of the z parameter on the quality of the results and on the computation time when indexing and querying 31-mers through their s-mers. "*constr.*" stands for "*construction*" and "*overest.*" stands for "*overestimation*". Results with $z = 0$ are equivalent to those obtained with the original `cBF` results. The Incorrect abundance calls are computed over true positive calls only.

is increased up to almost 100%. With $z = 20$, as we use $k = 31$, the size of the indexed s-mers is $s = 11$. The quasi-random existence of all s-mers of this size generates a huge amount of construction false positives, as seen in the last column. This has an effect on the running time, which is close to the query time of $z = 0$, undoubtedly because all s-mers queried are positives, annihilating the s-mer skipping optimization.

Caveat: in Table 4.2, the overestimations are reported only for true positive calls. Overall, also taking into account the negative answers, the overestimation rate with `fimpera` would be 0.03 with $z = 3$ for instance.

4.7.9 Query time

As mentioned in section 4.3.5, the `fimpera` approach does not increase the query execution time. On the contrary, it makes it possible to reduce the running time slightly as z increases. See Table 4.2.

4.7.10 Fixing s and varying k

In some setup (*e.g.* when k is not known during the indexation), it can be useful to index s-mers, and then choose k at query time. This allows the user to choose any k value ($\geq s$) even after the indexation step. Compared to having s and k fixed at indexing time, the major difference of this approach is that, since k is unknown when indexing, we cannot compute the s_{ab} of s-mers. This is because computing the s_{ab} of s-mers requires knowing the k value. Thus, in this setup, we rely on the abundance of s-mers instead of their s_{ab}.

In Table 4.3, we show that the false positive rate drops with regard to z, **for a given s value**.

Recall we are interested in indexing k-mers present at least twice, as stated in Section 4.7.3. As previously mentioned, when choosing k at query time, one cannot compute the s_{ab} of the s-mers. Hence, in this setup, the abundance of s-mers is higher than the situation when k and z are known at indexing time, in which one would have used the s_{ab}. Consequently, less s-mers have an abundance strictly lower than 2 and are filtered out, slightly increasing the FP rate comparatively with the setup in which k is fixed.

Note that choosing z at query time allows using a high value, such as 35, while still limiting the number of construction false positives. This is because the construction false positive rate mainly depends on the s value.

For $z = 3$, using the s_{ab}, `fimpera` reduces the false positive rate of the `cAMQ` filter by a factor 16 but slightly increases the number of overestimated calls. Recall that, in the Table 4.3, incorrect abundance calls are defined on true positive calls only. As such, for $z = 3$, the total proportion of incorrect calls of `fimpera` is $0.0507+(1-0.01507)*0.03248 = 0.0471 = 4.71\%$, while for the `cAMQ` filter it is 36.78%. For $z = 3$, compared to using the s_{ab}, using the abundance of s-mers yields 3 times more false positives and incorrect abundance calls. As such, whenever possible, using the s_{ab} is encouraged. We provide a comparison of the impact of choosing between the abundance of s-mers and the s_{ab} below.

z	0	1	2	3	4	5	6	7	8	35
False positive rate (%)	34.972	12.236	4.286	1.507	0.536	0.197	0.079	0.039	0.025	0.031
Of which: construction FP (%)	0	0.007	0.036	0.159	0.584	2.126	6.546	15.768	28.236	89.269
Incorrect abundance calls (%)	2.786	1.362	2.095	3.248	4.542	5.803	7.122	8.535	9.972	55.495
Of which: constr. overest. (%)	0	65.114	83.518	90.079	93.296	94.751	95.644	96.290	96.841	99.692

Table 4.3: Influence of the z parameter on the quality of the results when s if fixed. "*constr.*" stands for "*construction*" and "*overest.*" stands for "*overestimation*". The Incorrect abundance calls are computed over true positive calls only.

4.7.11 Comparison between s_{ab} and abundance of s-mers

To measure overestimations achieved when indexing the abundance of s-mers instead of their s_{ab}, we show overestimation results computed on exact abundances (not their *log* values) stored using 8 bits (and thus bounded to 255). This is motivated by the fact that, else, no difference could be seen. We used $k = 31$ and $s = 28$.

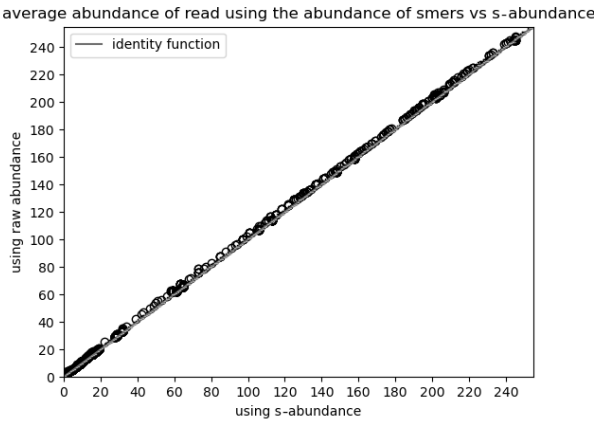


Figure 4.5: Comparison between the average abundance of k-mers from reads query results.

In Fig. 4.5, we show that overestimations introduced by the use of the abundance of each s-mer instead of s_{ab} are limited. This can be numerically estimated:

- the overestimation score comparing a raw `cAMQ` filter to the ground truth is $\approx 8.6 \times 10^7$;
- the overestimation score comparing `fimpera` to the ground truth is $\approx 10^4$ (three order of magnitude smaller);
- the additional overestimation score comparing the indexing of the abundance of s-mers instead of their $s - abundance$ with `fimpera` is $\approx 2 \times 10^4$ which appears negligible.

Extremely counter-intuitive case: the abundance of a s-mer can be smaller than its s_{ab}

Actually, in some rare setups, the abundance of a s-mer can be smaller than its s_{ab}, and even lower than the abundance of any k-mer that contains it (which would lead to an underestimation). This is only possible when indexing and querying canonical k-mers. This case happens with s even, when two k-mers having the same canonical form overlap over s characters. On a sequence of size $k+1$ composed of such overlap of two k-mers having the same canonical form, their canonical abundance in this sequence is 2, whereas the s-mer being the suffix of the first k-mer and the prefix of the second has an abundance of 1, thus smaller than its $s_{ab} = max(2) = 2$.

For instance, consider the sequence $seq = TACGTA$ and $k = 5, s = 4$. The first k-mer is $TACGT$ and the second k-mer is $ACGTA$. Both have the same canonical form $ACGTA$, whose abundance is then equal to two. The s-mer they share, $ACGT$, exists only once in seq, so its abundance is one, lower than its s_{ab} equal to two as provided by the abundance of the unique (in this situation) abundance of the canonical k-mer it belongs to.

This setup is rare (541 out of 281032928 s-mers of the indexed TARA dataset, i.e.: 0.000193 % of its s-mers). In our tests, this led to zero underestimation of the average of any read. Even if using the abundance of s-mers instead of their s_{ab}, and using their canonical versions breaks the theoretical absence of underestimation, we consider the biological impact of these underestimations as perfectly negligible.

4.7.12 Comparing `fimpera` and a counting Bloom filter with multiple hash functions

As stated in Section 4.3.1, the `fimpera` strategy may seem similar to associating multiple hash functions to a k-mer through its constituent s-mers, albeit two consecutive k-mers share z s-mers, allowing to not query the same s-mer multiple times.

In this section, we compared the reported abundance when indexing k-mers through $z + 1$ s-mers compared to indexing k-mers through $z + 1$ independent hash functions.

Using the overestimation score defined in Section 4.2.3 of the main manuscript, it is possible to compare `fimpera` ($z = 3$) using one hash function (ie. using 4 s-mers per k-mers) and using 4 independent hash functions. Using the same memory budget, $z = 3$ allows attaining an overestimation score of 98169 while using 4 independent hash functions allows reaching an overestimation score of 57967168 (the lower the better). The false positive rate attained by using $z = 3$ is $\approx 0.36\%$ while using 4 independent hash functions leads to a drastic increase of the false positive rate to 99.94%. This is due to the fact that using 4 independent hash functions saturates the filter and most absent k-mers calls lead to false positives.

4.8 Sliding window minimum benchmark

The sliding window minimum algorithm we propose is of independent interest. In this section, we propose a benchmark of the following algorithms to compare algorithms that

find the minimal value of each sliding window on a vector of integers v:

- "*recomputing*": a naive algorithm, computing naively the minimum in each window. Time: $\mathcal{O}(size\_window \times |v|)$.

- "*recomputing from last min*": keeping a reference to the minimum m in the previous window and recomputing the minimum in the current window only if the element of value m was the first element of the previous sliding window. Simply update m if the new element in the current window is smaller than m. Time: $\mathcal{O}(size\_window \times |v|)$ at worst (if v is an increasing sequence).

- a deque-based approach: the deque contains elements of the input vector. For each element e, the back of the deque is removed if it is strictly smaller than e. Then e is added on the back of the deque and elements that are out of the current window are removed. The minimum of the current window is then the front of the deque. Time: $\mathcal{O}(|v|)$, but requires time consuming memory allocations.

- "*fixed windows*": the approach described in Section 4.5, but allocating new vectors. Time: $\mathcal{O}(|v|)$, but requires time consuming memory heap allocations.

- Our proposal: "***fixed windows in place***": based on the fixed approach we also introduced (previous item), but without heap allocation, as described in algorithm 5. Time: $\mathcal{O}(|v|)$.

We propose various tests, highlighting the behaviors of the presented algorithms in different contexts: v is made up of random integers with either a small (size 9) or a large (size 100) window, and v is made up of increasing integers. In all cases, our proposal is the fastest algorithm. Recall that the implementation, of independent interest, is available at `https://github.com/lrobidou/sliding-minimum-windows` along with a recipe to reproduce and extend this benchmark.

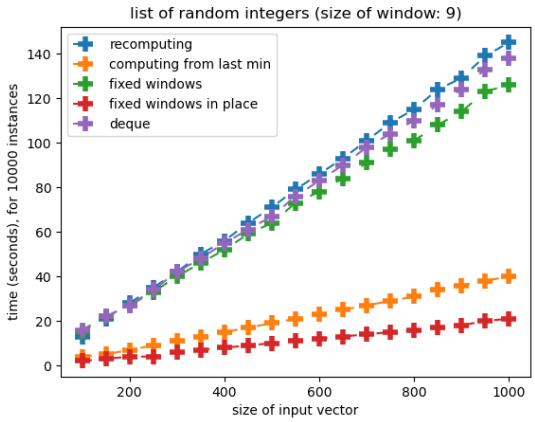


Figure 4.6: Comparison of five approaches for computing sliding window minimum. The input vector is composed of random integers. The size of the window is set to 9 and for each size of the input vector, 10000 instances were tested.

As shown in Fig. 4.6, when *size_window* is set to 9, approaches can be divided into two groups: those with heap allocations and those without. The fixed window approach with heap allocations is slower than keeping the last minimum ($\mathcal{O}(size\_window \times |v|)$ at worst), but once heap allocation is prevented (our proposal), this outperforms any other implementation, including keeping the last minimum by a factor 2. Heap allocations are costly: the differences between approaches in $\mathcal{O}(size\_window \times |v|)$ with heap allocation and in $\mathcal{O}(|v|)$ without them are negligible in practice (as long as $|v|$ is small enough, see Fig. 4.7).

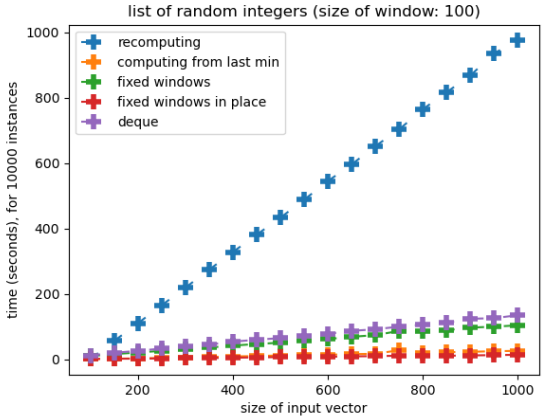


Figure 4.7: Comparison between 5 approaches for computing sliding window minimum. The input vector is composed of random integers. The size of the window is set to 100.

Results presented Figure 4.7 show that the naive approach in $\mathcal{O}(size\_window \times |v|)$ is significantly slower than other approaches, even if it does not require heap allocation.

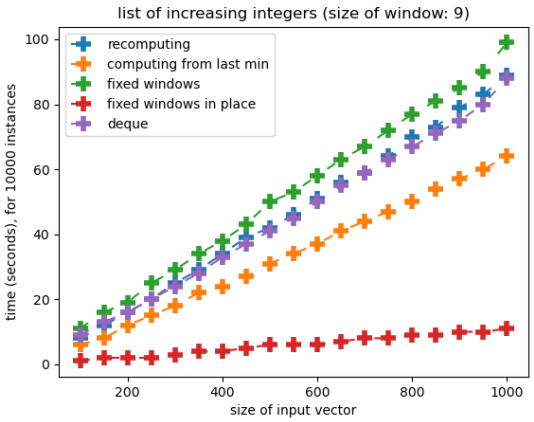


Figure 4.8: Comparison between 5 approaches for computing sliding window minimum. The size of the window is set to 9 and the input consists of an array of increasing values.

Keeping a reference to the minimum from the previous window is simpler than using a fixed window but it is two times slower. Moreover, if the input vector contains increasing values, it requires recomputing every window. Indeed, in this case, the minimum from any window is its leftmost element, and as it is out of the next window, the next window needs to be recomputed. See Fig. 4.8, to see the impact on run time. As illustrated in the

figure, the strategy of keeping a reference to the minimum of each window is slower than when applied to random values. In any case, the query time of the method we propose is constant regardless of the input vector's content.

4.9 Conclusion

We presented `fimpera`, a novel computational method for reducing the false positive rate and increasing precision in any counting Approximate Membership Query data structure. This is achieved without requiring any changes to the original data structure, with no memory overhead, and even with a slight improvement in query computation time.

Our results showed that when applied on top of a counting Bloom filter, `fimpera` enabled to yield 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 couting Approximate Membership Query filter, `fimpera` requires setting up a unique parameter, z. Fortunately, results are highly robust with the choice of z, unless extreme values are chosen.

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.

Finally, of independent interest, we proposed a novel algorithm and its implementation for computing the minimal or maximal values of consecutive windows, sliding on an array of integers or floats. To the best of our knowledge, this is the fastest algorithm to perform this task.

This work has been added to kmindex [6], a method for indexing genomic samples through their k-mers'count. `fimpera` allowed kmindex to scale to 1393 complex metagenome samples of sequences from the *Tara* Oceans project.

Conclusion

In this work, we proposed methods to reduce the false positive rate and the overestimation rate of both `AMQ` and `cAMQ` filters, with no practical drawbacks when used with the recommended parameter. Alternatively, for a given false positive rate, these methods allow reducing the size used by a (counting) Bloom filter by an order of magnitude. No matter the tradeoff precision/size, our contributions allow to reduce the query time of the underlying data structure.

These methods have been merged into kmindex, a method for indexing and querying genomic samples. `findere` allowed kmindex to index all the *Tara* oceans metagenomic datasets. With `findere`, indexing 50 of the *Tara* metagenomic datasets requires 164 GB. Without `findere`, this would require more than one order of magnitude more memory, which is prohibitive.

Of independent interest, we also proposed an algorithm for computing the minimum of fixed-size sliding windows of a vector of integers. We provide an implementation as well as a benchmark comparing it to other approaches.

Each of our implementations is publicly available as a free software.

We are expecting an important impact of the `findere` and `fimpera` tools. Indeed, `AMQ` data structures are essential for indexing large datasets. In particular, their usage is fundamental for indexing the genomic sequencing data, for which they offer a new scaling breakthrough.

In this section, I will present discussions on multiple levels. First, I will focus on the technical point of view. Then, I will focus on the improvement that can be done in the near and long-term future. Finally, I will discuss the environmental impact of bioinformatics.

4.10 Technical discussions

4.10.1 Compression

Currently, the implementations of the indexes we propose are not compressed. Compressing these indexes would be beneficial to help face the data deluge on bioinformatics. Since

it consists of random bits set to *true*, a Bloom filter is likely to be hardly compressible. However, merging them in a COBS-like fashion introduces a pattern in the data structure: if a k-mer is present in multiple datasets, then the region of the data structure holding this information will have multiple (possibly contiguous) bits set to *true*. Compressing a filter implies the need to decompress (parts of) it at query time. As such, compression may slow down the query of s-mers. To mitigate this issue, the compression scheme should allow fast random access, like the LZ-End [67] scheme.

4.10.2 Reducing the size of the datastructure

Using the currently available datastructures, indexing the sheer number of datasets publicly available would lead to petabytes-scale indexes. To mitigate this issue, some methods subsamples k-mers, like Mash [68] (*e.g.* indexing k-mers whose hash value is lower than a threshold, says t). This allows computing the distance between two datasets, but due to the sampling of k-mers, information for a single k-mer cannot be retrieved.

Inspired by Mash, I propose to use a deterministic algorithm to subsamples s-mers (*e.g.* using their hash values). At indexation time, multiple thresholds could be used, partitioning indexed s-mers into multiple buckets according to their hash value. s-mers having a high hash value (higher than the highest threshold) would be quickly queryable, while the others would not. For instance, the first bucket (associated with the highest threshold) would not be compressed, the second compressed with fast random access, the third highly compressed, the fourth highly compressed on external memory, etc.

At query time, one could query the s-mers belonging to the first partition. Absent s-mers from this partition would indicate the absence of some k-mers in the query. s-mers appearing only in absent k-mers would not be marked as "unnecessary". The query would continue by querying s-mers belonging to the second bucket (except for the "unnecessary" s-mers). This process would continue until all the k-mers have been answered.

This would allow space gain by compressing some filters and would lower the need for RAM, as some filters could be placed on disk. The impact on query time would depend on the number of s-mers in the first partitions (quickly queryable).

4.10.3 Replacing k-mers

Recall that the reason we use k-mers is to compare sequences without alignment. However, using spaced seed yields better similarity estimations in the context of metagenomic [69].

In a nutshell, spaced seeds are "k-mers with mismatch allowed". However, unlike k-mers, they cannot be trivially split into s-mers. The same goes for strobemers [70], an alternative to k-mers allowing matching similar sequences, even when their similar regions contain insertion or deletion of bases compared to one another. In a nutshell, strobemers consist of n shorter l-mers, called strobes. Strobes are subsamples of a sequence, and the space between two strobes in that sequence is variable.

Splitting strobemers into s-mers may allow a better similarity estimation when using our indexes. However, it is probably going to be non-trivial, as two consecutive strobemers do not overlap. A direction I am interested in is: instead of splitting strobemers into s-mers, I would rather consider strobemers as s-mers and coin a new concept consisting of $z+1$ strobemers. A window of $z+1$ consecutive strobemers would be considered as present if and only if the $z+1$ strobemers in this window are present.

4.11 Environmental impact of bioinformatics

As stated in the introduction of this manuscript, "the ability to sequence DNA is currently having a transformative impact on many scientific fields". Research also has impacts on our society. However, I could not find many manuscripts actually discussing these impacts. I propose to quickly address this lack here, by discussing one negative impact of computer science on our environment.

A study [71] analyzed the production of CO_2 by bioinformatics tools. A metagenomic assembler, metaSPAdes [72], was run on 100 samples from forest soil. This emitted 186 $kgCO_2eq$ in 29 hours, which is equivalent to what is emitted when driving 1 065 km by a car in Europe. A tree would require 203 months to absorb this much CO_2.

Because some countries rely on low-carbon power (*e.g.* nuclear energy in France) while the USA relies on coil and gas to produce electricity, the environmental cost of an algorithm depends on where it is run.

A researcher once said during a workshop that their indexation methods emitted as much CO_2 as a trip by plane from France to the USA. If true, this means that if your biological dataset is stored in a US cluster, then taking a plane to get the hard drive that contains your data, going back to Europe, and performing your computation at home could emit less CO_2 than running the computation in the cluster.

Nowadays, people are increasingly aware that taking planes emits CO_2. However, it seems to me that very few people think about CO_2 emissions when they press "enter" in

their terminal. Personally, I have no idea how much CO_2 my Ph.D. emitted (*e.g.* through my CI pipeline or my computer components).

Not understanding what emits CO_2 limits our ability to reduce those emissions. As such, a French research group proposes to measure the CO_2 emission of research labs (`https://labos1point5.org/`). Doing the same with the emission of computational resources would allow understanding how to reduce them. For instance, approaches like `findere` and `fimpera` could reduce the carbon footprint of queries, as they require less query time and less disk space.

BIBLIOGRAPHY

1. Sanger, F. & Coulson, A. R., A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase, *Journal of molecular biology* **94,** 441–448 (1975).
2. Margulies, M. *et al.,* Genome sequencing in microfabricated high-density picolitre reactors, *Nature* **437,** 376–380 (2005).
3. Integrative, H. *et al.,* The integrative human microbiome project, *Nature* **569,** 641–648 (2019).
4. Lemane, T., Medvedev, P., Chikhi, R. & Peterlongo, P., Kmtricks: efficient and flexible construction of bloom filters for large sequencing data collections, *Bioinformatics Advances* **2,** vbac029 (2022).
5. Sunagawa, S. *et al.,* Tara Oceans: towards global ocean ecosystems biology, *Nature Reviews Microbiology* **18,** 428–445 (2020).
6. Lemane, T. *et al.,* kmindex and ORA: indexing and real-time user-friendly queries in terabytes-sized complex genomic datasets, *bioRxiv,* 2023–05 (2023).
7. Fogarty, H. E., Burrows, M. T., Pecl, G. T., Robinson, L. M. & Poloczanska, E. S., Are fish outside their usual ranges early indicators of climate-driven range shifts?, *Global Change Biology* **23,** 2047–2057 (2017).
8. Masters, P. S., The molecular biology of coronaviruses, *Advances in virus research* **66,** 193–292 (2006).
9. Khan, S. *et al.,* Coronavirus diseases 2019: current biological situation and potential therapeutic perspective, *European Journal of Pharmacology* **886,** 173447 (2020).
10. `https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid`.
11. Palomaki, G. E. *et al.,* DNA sequencing of maternal plasma to detect Down syndrome: an international clinical validation study, *Genetics in medicine* **13,** 913–920 (2011).
12. `https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000037856379`.
13. de-Dios, T. *et al.,* Metagenomic analysis of a blood stain from the French revolutionary Jean-Paul Marat (1743–1793), *Infection, Genetics and Evolution* **80,** 104209, ISSN: 1567-1348, `https://www.sciencedirect.com/science/article/pii/S1567134820300411` (2020).

14. Břinda, K. *et al.,* Rapid inference of antibiotic resistance and susceptibility by genomic neighbour typing, *Nature microbiology* **5,** 455–464 (2020).

15. Børsting, C. & Morling, N., Next generation sequencing and its applications in forensic genetics, *Forensic Science International: Genetics* **18,** 78–89 (2015).

16. Yano, K. *et al.,* Genome-wide association study using whole-genome sequencing rapidly identifies new genes influencing agronomic traits in rice, *Nature genetics* **48,** 927–934 (2016).

17. Pierella Karlusich, J. J., Ibarbalz, F. M. & Bowler, C., Phytoplankton in the Tara ocean, *Annual Review of Marine Science* **12,** 233–265 (2020).

18. Wu, R., Nucleotide sequence analysis of DNA: I. Partial sequence of the cohesive ends of bacteriophage λ and 186 DNA, *Journal of molecular biology* **51,** 501–521 (1970).

19. Maxam, A. M. & Gilbert, W., A new method for sequencing DNA. *Proceedings of the National Academy of Sciences* **74,** 560–564 (1977).

20. Shendure, J. & Ji, H., Next-generation DNA sequencing, *Nature biotechnology* **26,** 1135–1145 (2008).

21. Eid, J. *et al.,* Real-time DNA sequencing from single polymerase molecules, *Science* **323,** 133–138 (2009).

22. Haque, F., Li, J., Wu, H.-C., Liang, X.-J. & Guo, P., Solid-state and biological nanopore for real-time sensing of single chemical and sequencing of DNA, *Nano today* **8,** 56–74 (2013).

23. Sereika, M. *et al.,* Oxford Nanopore R10. 4 long-read sequencing enables the generation of near-finished bacterial genomes from pure cultures and metagenomes without short-read or reference polishing, *Nature methods* **19,** 823–826 (2022).

24. Amid, C. *et al.,* The European Nucleotide Archive in 2019, *Nucleic Acids Research* **48,** D70–D76, ISSN: 0305-1048, `https://doi.org/10.1093/nar/gkz1063` (2021) (Jan. 2020).

25. Bloom, B. H., Space/time trade-offs in hash coding with allowable errors, *Communications of the ACM* **13,** 422–426 (1970).

26. Fan, L., Cao, P., Almeida, J. & Broder, A. Z., Summary cache: a scalable wide-area web cache sharing protocol, *IEEE/ACM transactions on networking* **8,** 281–293 (2000).

27. Robidou, L. & Peterlongo, P., *findere: fast and precise approximate membership query in String Processing and Information Retrieval: 28th International Symposium, SPIRE 2021, Lille, France, October 4–6, 2021, Proceedings 28* (2021), 151–163.

28. Robidou, L. & Peterlongo, P., fimpera: drastic improvement of Approximate Membership Query data-structures with counts, *Bioinformatics* **39,** btad305 (2023).

29. Pellow, D., Filippova, D. & Kingsford, C., Improving Bloom Filter Performance on Sequence Data Using k -mer Bloom Filters, en, *Journal of Computational Biology* **24,** 547–557, ISSN: 1557-8666, http://www.liebertpub.com/doi/10.1089/cmb.2016.0155 (2021) (June 2017).

30. Roberts, M., Hayes, W., Hunt, B. R., Mount, S. M. & Yorke, J. A., Reducing storage requirements for biological sequence comparison, *Bioinformatics* **20,** 3363–3369 (2004).

31. Li, Y. *et al.,* MSPKmerCounter: a fast and memory efficient approach for k-mer counting, *arXiv preprint arXiv:1505.06550* (2015).

32. Limasset, A., Rizk, G., Chikhi, R. & Peterlongo, P., Fast and scalable minimal perfect hashing for massive key sets, *arXiv preprint arXiv:1702.03154* (2017).

33. Pibiri, G. E. & Trani, R., *PTHash: Revisiting FCH minimal perfect hashing in Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval* (2021), 1339–1348.

34. Pibiri, G. E., Shibuya, Y. & Limasset, A., Locality-preserving minimal perfect hashing of k-mers, *Bioinformatics* **39,** i534–i543 (2023).

35. Rahman, A. & Medevedev, P., Representation of k-mer sets using spectrum-preserving string sets, *Journal of Computational Biology* **28,** 381–394 (2021).

36. De Bruijn, N. G., A combinatorial problem, *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam* **49,** 758–764 (1946).

37. Břinda, K., Baym, M. & Kucherov, G., Simplitigs as an efficient and scalable representation of de Bruijn graphs, *Genome biology* **22,** 1–24 (2021).

38. Schmidt, S. & Alanko, J. N., Eulertigs: minimum plain text representation of k-mer sets without repetitions in linear time, *bioRxiv,* 2022–05 (2022).

39. Marchet, C., Iqbal, Z., Gautheret, D., Salson, M. & Chikhi, R., REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets, *Bioinformatics* **36,** i177–i185 (2020).

40. Marchet, C. *et al.,* Data structures based on k-mers for querying large collections of sequencing data sets, *Genome Research* **31,** 1–12 (2021).

41. Chikhi, R., Holub, J. & Medvedev, P., Data structures to represent a set of k-long DNA sequences, *ACM Computing Surveys (CSUR)* **54,** 1–22 (2021).

42. Almodaresi, F., Sarkar, H., Srivastava, A. & Patro, R., A space and time-efficient index for the compacted colored de Bruijn graph, *Bioinformatics* **34,** i169–i177 (2018).

43. Marchet, C., Kerbiriou, M. & Limasset, A., BLight: efficient exact associative structure for k-mers, *Bioinformatics* **37,** 2858–2865 (2021).

44. Holley, G. & Melsted, P., Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs, *Genome biology* **21,** 1–20 (2020).

45. Pibiri, G. E., Sparse and skew hashing of k-mers, *Bioinformatics* **38,** i185–i194 (2022).

46. Karasikov, M. *et al.,* Metagraph: Indexing and analysing nucleotide archives at petabase-scale, *BioRxiv,* 2020–10 (2020).

47. Almeida, A. *et al.,* A unified catalog of 204,938 reference genomes from the human gut microbiome, *Nature biotechnology* **39,** 105–114 (2021).

48. O'Leary, N. A. *et al.,* Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation, *Nucleic acids research* **44,** D733–D745 (2016).

49. Zhang, X., Yu, Y., Mok, C. H., MacLeod, J. N. & Liu, J., *Gazelle: transcript abundance query against large-scale RNA-seq experiments in Proceedings of the 12th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics* (2021), 1–8.

50. Carter, L., Floyd, R., Gill, J., Markowsky, G. & Wegman, M., *Exact and approximate membership testers in Proceedings of the tenth annual ACM symposium on Theory of computing* (1978), 59–65.

51. Cormode, G. & Muthukrishnan, S., An improved data stream summary: the count-min sketch and its applications, *Journal of Algorithms* **55,** 58–75 (2005).

52. Bender, M. A. *et al., Don't thrash: how to cache your hash on flash in 3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)* (2011).

53. Pandey, P., Bender, M. A., Johnson, R. & Patro, R., *A general-purpose counting filter: Making every bit count in Proceedings of the 2017 ACM international conference on Management of Data* (2017), 775–787.

54. Fan, B., Andersen, D. G., Kaminsky, M. & Mitzenmacher, M. D., *Cuckoo filter: Practically better than bloom in Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014), 75–88.

55. Graf, T. M. & Lemire, D., Xor filters: Faster and smaller than bloom and cuckoo filters, *Journal of Experimental Algorithmics (JEA)* **25,** 1–16 (2020).

56. Graf, T. M. & Lemire, D., Binary fuse filters: Fast and smaller than xor filters, *Journal of Experimental Algorithmics (JEA)* **27,** 1–15 (2022).

57. Bradley, P., Den Bakker, H. C., Rocha, E. P., McVean, G. & Iqbal, Z., Ultrafast search of all deposited bacterial and viral genomic data, *Nature biotechnology* **37,** 152–159 (2019).

58. Wong, H. K., Bit transposed files (1985).

59. Bingmann, T., Bradley, P., Gauger, F. & Iqbal, Z., *COBS: a compact bit-sliced signature index in String Processing and Information Retrieval: 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7–9, 2019, Proceedings 26* (2019), 285–303.

60. Solomon, B. & Kingsford, C., Fast search of thousands of short-read sequencing experiments, *Nature biotechnology* **34,** 300–302 (2016).

61. Harris, R. S. & Medvedev, P., Improved representation of sequence bloom trees, *Bioinformatics* **36,** 721–727 (2020).

62. Marchet, C. & Limasset, A., Scalable sequence database search using Partitioned Aggregated Bloom Comb-Trees, *bioRxiv,* 2022–02 (2022).

63. Darvish, M., Seiler, E., Mehringer, S., Rahn, R. & Reinert, K., Needle: a fast and space-efficient prefilter for estimating the quantification of very large collections of expression experiments, *Bioinformatics* **38,** 4100–4108 (2022).

64. Lee, H., Shuaibi, A., Bell, J. M., Pavlichin, D. S. & Ji, H. P., Unique k-mer sequences for validating cancer-related substitution, insertion and deletion mutations, *NAR cancer* **2,** zcaa034 (2020).

65. Kokot, M., Długosz, M. & Deorowicz, S., KMC 3: counting and manipulating k-mer statistics, *Bioinformatics* **33,** 2759–2761 (2017).

66. Durocher, S. & Singh, R., A simple linear-space data structure for constant-time range minimum query, *Theoretical Computer Science* **770,** 51–61 (2019).

67. Kreft, S. & Navarro, G., *LZ77-like compression with fast random access in 2010 Data Compression Conference* (2010), 239–248.

68. Ondov, B. D. *et al.,* Mash: fast genome and metagenome distance estimation using MinHash, *Genome biology* **17,** 1–14 (2016).

69. Břinda, K., Sykulski, M. & Kucherov, G., Spaced seeds improve k-mer-based metagenomic classification, *Bioinformatics* **31,** 3584–3592 (2015).

70. Sahlin, K., Effective sequence similarity detection with strobemers, *Genome research* **31,** 2080–2094 (2021).

71. Grealey, J. *et al.,* The carbon footprint of bioinformatics, *Molecular biology and evolution* **39,** msac034 (2022).

72. Nurk, S., Meleshko, D., Korobeynikov, A. & Pevzner, P. A., metaSPAdes: a new versatile metagenomic assembler, *Genome research* **27,** 824–834 (2017).

73. Holley, R. W., Madison, J. T. & Zamir, A., A new method for sequence determination of large oligonucleotides, *Biochemical and Biophysical Research Communications* **17,** 389–394 (1964).

74. Twiss, P., Hill, M., Daley, R. & Chitty, L. S., *Non-invasive prenatal testing for Down syndrome in Seminars in Fetal and Neonatal Medicine* **19** (2014), 9–14.

75. Bray, N. L., Pimentel, H., Melsted, P. & Pachter, L., Near-optimal probabilistic RNA-seq quantification, *Nature biotechnology* **34,** 525–527 (2016).

76. Almodaresi, F., Sarkar, H., Srivastava, A. & Patro, R., A space and time-efficient index for the compacted colored de Bruijn graph, *Bioinformatics* **34,** i169–i177, ISSN: 1367-4803, eprint: `https://academic.oup.com/bioinformatics/article-pdf/34/13/i169/50316242/bioinformatics\_34\_13\_i169.pdf`, `https://doi.org/10.1093/bioinformatics/bty292` (June 2018).

77. Chikhi, R., Limasset, A. & Medvedev, P., Compacting de Bruijn graphs from sequencing data quickly and in low memory, *Bioinformatics* **32,** i201–i208 (2016).

78. Italiano, G. F., Prezza, N., Sinaimeri, B. & Venturini, R., *Compressed weighted de Bruijn graphs in CPM 2021-32nd Annual Symposium on Combinatorial Pattern Matching* **191** (2021), 1–16.

79. Bowe, A., Onodera, T., Sadakane, K. & Shibuya, T., *Succinct de Bruijn graphs in Algorithms in Bioinformatics: 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings 12* (2012), 225–235.

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Moteur de recherche pour données de séquençage génomique

Mot clés : requêtes d'appartenance approchée, structure de données, indexation, k-mers, filtres

Résumé : Les technologies de séquençage à haut débit génèrent des quantités massives de jeux de données de séquences biologiques à mesure que les coûts diminuent. L'un des défis actuels pour exploiter ces données consiste à développer des moteurs de recherche pour ces jeux d'une taille de l'ordre du pétaoctet. La plupart des méthodes existantes reposent sur l'indexation des séquences via leurs mots de longueur k, appelés k-mers. Dans de nombreux domaines de la bioinformatique, il est nécessaire de retrouver l'abondance d'un k-mer dans un ensemble de données.

Des structures de données, appelées *AMQ*, sont largement utilisées pour représenter ces grands ensembles de k-mers. D'autres structures de données similaires, les *cAMQ*, représentent des multiensembles, de façon à pouvoir retrouver l'abondance d'un k-mer dans un jeu. Cependant, par nature, ces AMQs renvoient des faux positifs et, dans le cas d'un multiensemble, ont tendance à surestimer l'abondance des k-mers.

Dans ce manuscrit, nous présentons deux contributions, `findere` et `fimpera`, qui permettent d'améliorer les performances des (c)AMQs. Appliqué au filtre de Bloom, qui est largement utilisé en bioinformatique, `findere` réduit son taux de faux positifs de deux ordres de grandeur tout en accélérant ses requêtes. `fimpera` réduit le taux de faux positifs d'un filtre de Bloom avec comptage tout en améliorant la précision des abondances renvoyées.

Title: Search engine for genomic sequencing data

Keywords: approximate membership query, data structures, indexation, k-mers, filters

Abstract: High throughput sequencing technologies generate massive amounts of biological sequence datasets as costs fall. One of the current algorithmic challenges for exploiting these data on a global scale consists in providing efficient query engines on these petabyte-scale datasets. Most methods indexing those datasets rely on indexing sequence datasets through their constituent words of fixed length k, called k-mers. In many applications, it is required to retrieve the abundance of a k-mer in a dataset.

Approximate membership query (AMQ) data structures are widely used for representing these large sets of k-mers. A counting AMQ (cAMQ) can index the abundance of k-mers along with their presence in a dataset. However, these (c)AMQs suffer by nature from non-avoidable false-positive calls and overestimated calls that bias downstream analyses.

In this work, we propose two contributions, `findere` and `fimpera`, that enable the improvement of any (c)AMQ performance. Applied to the widely used Bloom filter, `findere` reduces the false positive rate by two orders of magnitude while speeding up the queries. `fimpera` reduces the false positive rate of a counting Bloom filter while improving the precision of the reported abundances.